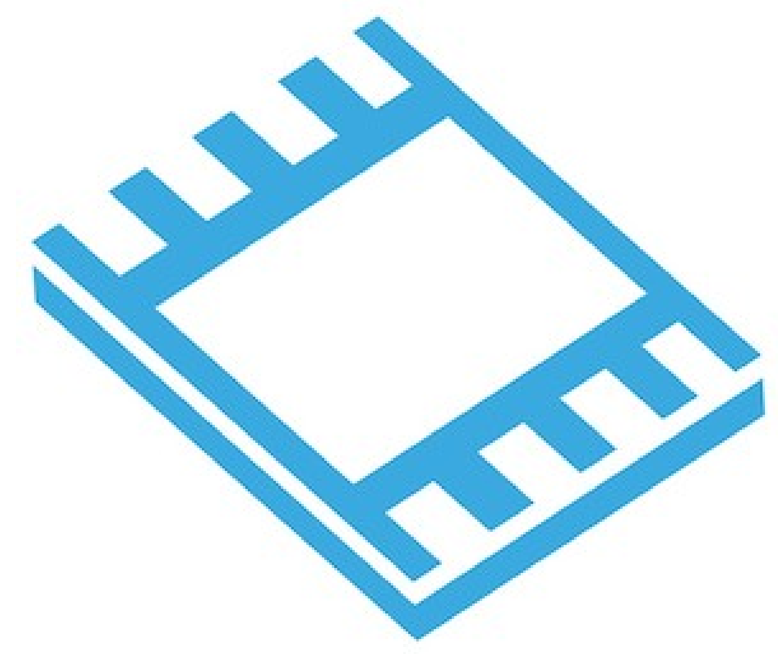
МІКРОПРОЦЕСОРИ В РАДІОСИСТЕМАХ І ПРИСТРОЯХ: ПОЧАТОК РОБОТИ З МІКРОКОНТРОЛЕРАМИ STM32



Міністерство освіти і науки України
Чернівецький національний університет
імені Юрія Федьковича

МІКРОПРОЦЕСОРИ В РАДІОСИСТЕМАХ І ПРИСТРОЯХ: ПОЧАТОК РОБОТИ З МІКРОКОНТРОЛЕРАМИ STM32

Навчальний посібник

Укладачі: А.Д. Верига, Р.Л. Політанський,
О.В. Круліковський, Г.В. Косован

Чернівці
Чернівецький національний університет
імені Юрія Федьковича
2023

УДК 004.4(076.5)
М 784

Рекомендовано до друку вченою радою навчально-наукового
Інституту фізико-технічних та комп'ютерних наук
(протокол № 7 від 22 вересня 2023 року)

М 784

Мікропроцесори в радіосистемах і пристроях: початок роботи з мікроконтролерами STM32 : навч. посіб. уклад. : Верига А. Д., Політанський Р. Л., Круліковський О.В., Косован Г.В. , Чернівці : Чернівец. нац. ун-т ім. Ю. Федьковича, 2023. 188 с.

У виданні розглянуті основні особливості 32-бітних мікроконтролерів STM32 з ядром Cortex-M, бібліотеки взаємодії програм, використовувані для програмування даних мікро контролерів. Висвітлено основи програмування мовою високого рівня Сі, найбільш популярні програмні середовища для написання програм і програмування. Наведено кілька прикладів програм для таких мікроконтролерів. Посібник відповідає навчальному плану дисципліни нормативної підготовки "Мікропроцесори в радіосистемах і пристроях". Для студентів магістратури спеціальності 172 – "Електронні комунікації та радіотехніка".

УДК 004.4(076.5)

© Чернівецький національний університет
імені Юрія Федьковича, 2023

ЗМІСТ

РОЗДІЛ 1. ОСНОВНІ ОСОБЛИВОСТІ МІКРОКОНТРОЛЕРІВ CORTEX STM32 7
1.1. Загальна інформація 7
1.2. Огляд Cortex 13
1.3. ЦПП Corte. 13
1.4. Процесор Cortex 26
1.5. Система тактування та синхронізації 32
1.6. Література до розділу 34
РОЗДІЛ 2. СТАНДАРТИ ПРОГРАМНОГО ІНТЕРФЕЙСУ МІКРОКОНТРОЛЕРІВ CORTEX ТА БІБЛІОТЕКИ ДЛЯ РОБОТИ З STM32 36
2.1. Бібліотеки взаємодії програм з ARM Cortex 36
2.2. Бібліотека CMSIS 39
2.3. Стандартна бібліотека периферії STM32 SPL 47
2.4. Бібліотека HAL та LL 53
2.5. Приклади програм 62
2.6. STM32 без HAL и SPL 63
2.7. Література до розділу 69
РОЗДІЛ 3. ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ ВИСОКОГО РІВНЯ СІ 70
3.1. Ввідні поняття 70
3.2. Структура програми мовою Сі 74
3.3. Типи даних, змінні, константи, структури 79
3.4. Вказівники та адреси змінних 86
3.5. Масиви і рядки 89
3.6. Оператори розгалудження 91
3.7. Циклічні конструкції 93
3.8. Директиви препроцесора 96
3.9. Управління регістрами мікроконтролера 100
3.10. Література до розділу 102
РОЗДІЛ 4. СЕРЕДОВИЩА РОЗРОБЛЕННЯ ПРОГРАМ ДЛЯ МІКРОКОНТРОЛЕРІВ STM32. ВСТАНОВЛЕННЯ, НАЛАШТУВАННЯ, ПОРЯДОК РОБОТИ, СТВОРЕННЯ ПРОЄКТУ 103
4.1. Середовища розроблення (IDE, Integrated Development Environment) і компілятори 104
4.2. Keil uVision 108
4.3. Програма початкових налаштувань STM32CubeMX 126
4.4. Програмне середовище розробки програм STM32CubeIDE 135
4.5. Початок роботи з STM32 і STM32CubeIDE 136

4.6. Створення нового STM32 проєкту 139
4.7. Приклад створення проєкту в середовищі STM32CubeIDE..........151
4.8. Література до розділу..158
РОЗДІЛ 5. ПРИКЛАДИ ПРОГРАМ ДЛЯ STM32.........................160
5.1. Використання портів введення/виведення....................................160
5.2. Переривання та їх використання. Використання таймерів............161
5.3. Генерація сигналу ШІМ...162
5.4. Використання АЦП..165
5.5. Використання USART 166
5.6. Робота з SPI...167
5.7. Робота з DMA 170
5.8. Використання аналогового компаратора 172
5.9. Операційний підсилювач...174
5.10. Література до розділу...176
ДОДАТКИ…………………………………………………………...…...178

Скорочення

ADC (Analog-to-digital converter) – аналогово-цифровий перетворювач (АЦП);

AHB (Advanced Highspeed Bus) – розширена високошвидкісна шина;

ALU (arithmetic logic unit) – арифметичний логічний пристрій (АЛП);

APB (Advanced Peripheral Bus) – розширена периферійна шина;

ARM (Advanced RISC Machine) – розширена машина RISC;

CAN (Controller Area Network) – локальна мережа контролерів;

DMA (Direct Memory Access) – прямий доступ до пам’яті (ПДП), режим обміну даними між пристроями або ж між пристроєм і основною пам'яттю без участі центрального процесора;

Embedded Trace Macrocell (ETM) – вбудована трасувальна мікрокомірка для забезпечення відстеження в реальному часі інструкцій і даних ARM946E-S у вбудованій системі;

FPB (Flash Patch and Breakpoint) – блок реалізації апаратних точок зупину та перенаправлення коду і даних з простору коду в простір системи;

IIC, I2C (Inter-Integrated Circuit) – послідовна шина даних для зв'язку інтегральних схем;

MCU – microcontroller unit;

NVIC (Nested vectored interrupt controller) – контролер вкладених переривannь;

PLL (phase-locked loop) – фазове автопідлаштування частоти (ФАПЧ), система керування, яка генерує вихідний сигнал, фаза якого пов’язана з фазою вхідного сигналу;

RISC (Reduced Instruction Set Computing) – обчислення зі скороченим набором команд, архітектура процесорів зі скороченим набором команд;

RTOS (Real time operational system) – операційна система реального часу;

SoC (System on crystall) – система на кристалі;

SPI (Serial Peripheral Interface) – послідовний периферійний інтерфейс;

SRAM (Static random access memory) – статична оперативна пам'ять;

SWV (Serial Wire Viewer) – функція трасування даних, яка є в багатьох процесорах Arm Cortex-M3, M4, M7, M23 і M33;

USB (Universal Serial Bus) – універсальна послідовна шина, стандарт роз'ємів і кабелів для передачі даних (до 40 Гбіт/с) та живлення (до 240 Вт) невеликих пристроїв;

ОС – операційна система;

ПЗ – програмне забезпечення;

ЦПП – центральний процесорний пристрій.

РОЗДІЛ 1. Основні особливості мікроконтролерів Cortex STM32

1.1. Загальна інформація

Процесори ARM є ключовим компонентом для великої кількості 32-бітних вбудованих систем. Процесори ARM широко використовуються в мобільних телефонах, планшетах, мікрокомп'ютерах та інших портативних пристроях. ARM засновані на RISC-архітектурі, що дозволяє зменшити споживання енергії процесором, отже, робить їх ідеальним вибором для вбудованих систем.

Хоча ARM основані на RISC-архітектурі, вони не повністю повторюють принципи побудови таких систем. Для того, щоб зробити ARM більш пристосованими до використання у вбудованих системах, довелося піти на такі відхилення від принципів RISC:

- змінна кількість циклів виконання для простих інструкцій. Прості інструкції ARM можуть виконуватися кілька циклів. Наприклад, виконання інструкцій Load і Save залежить від кількості регістрів, які їм передані;
- можливість поєднувати команди зсуву і обертання з командами обробки інформації;
- умовне виконання – інструкція виконується лише в тому випадку, якщо виконується конкретна умова, що збільшує продуктивність і дозволяє позбутися операторів розгалуження;
- поліпшені інструкції сигнального оброблення – процесори ARM підтримують поліпшені DSP-інструкції для операцій з цифровими сигналами.

Програміст може розглядати ядро ARM як набір функціональних блоків – ALU, MMU та ін., з'єднаних шиною даних. Дані надходять в процесор через шину даних, декодер інструкцій обробляє інструкції перед їх виконанням. ARM можуть працювати тільки з даними, які записані в регістрах. Тому перед виконанням інструкцій в регістри записуються дані для їх виконання. ALU зчитує дані з регістрів, виконує необхідні

операції і записує результат назад в регістр, звідки його можна записати в зовнішню пам'ять.

Сімейство ARM Cortex – це нове покоління процесорів зі стандартизованою архітектурою для виконання широкого кола технологічних завдань. На відміну від інших ядер ARM, сімейство Cortex є закінченим процесорним ядром зі стандартним 32-бітовим ЦПП і системною архітектурою, що включає систему шин, модуль вкладених переривань, систему відладки і стандартно розміщену пам'ять.

Сімейство Cortex має три основні профілі:

A – для високопродуктивних застосувань,

R – для додатків реального часу,

M – для бюджетних застосувань.

У STM32 використовується профіль Cortex-M3, розроблений спеціально для систем, що поєднують в собі високу продуктивність і низьке енергоспоживання. Ціна цього сімейства досить низька, щоб конкурувати з 8- і 16-розрядними мікроконтролерами.

Попри те, що ядра ARM7 і ARM9 були успішно інтегровані в стандартні мікроконтролери, в них простежується спадковість від систем на кристалі (SoC). Особливо це проявляється при обробці виняткових ситуацій і переривань, оскільки кожен виробник мікроконтролерів розробляє власне рішення. Cortex-M3 – це стандартизоване мікроконтролерне ядро, що виходить за межі ЦПП, що є закінченим серцем мікроконтролера (включає систему переривань, системний таймер, систему налагодження і карту пам'яті). Адресний простір 4 Гбайт Cortex-M3 розділений на строго певні області для коду додатка, SRAM, периферійних і системних пристроїв. На відміну від ARM7, Cortex-M3 побудований на базі Гарвардської архітектури пам'яті і, отже, містить безліч шин, що дозволяють виконувати операції паралельно, збільшуючи загальну продуктивність. В порівнянні з ранньою архітектурою ARM, сімейство Cortex може здійснювати доступ до невирівняних даних, що гарантує ефективніше використання вбудованої SRAM.

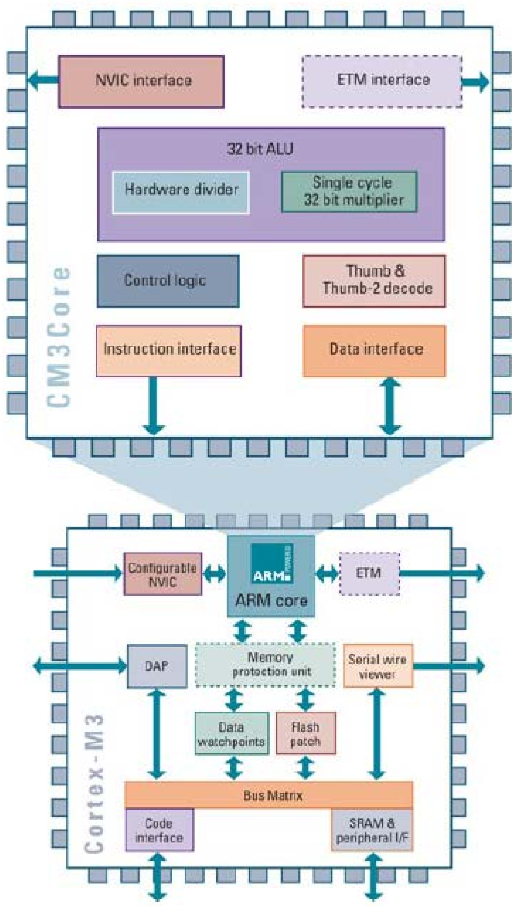


Рис. 1.1. Процесорне ядро Cortex-M3: NVIC interface – інтерфейс контролера вкладених переривань, ETM interface – ETM інтерфейс для забезпечення відстеження в реальному часі інструкцій і даних у вбудованій системі, 32 bit ALU – 32-бітове АЛП, Hardware divider – Апаратний дільник, Single cycle 32 bit multiplier – 32-бітовий одноцикловий помножувач, Control logic – Управляюча логіка, Thumb & Thumb-2 decode – декодер інструкцій Thumb і Thumb-2, Instruction interface – Інтерфейс команд, Data interface – Інтерфейс даних. Configurable NVIC – Конфігурований NVIC, ARM Core – Ядро ARM, Memory protection unit – Модуль захисту пам'яті, Serial wire viewer (SWV) – функція трасування даних, Data watchpoints – Точки контролю даних, Flash Patch and Breakpoint – блок реалізації апаратних точок зупину та перенаправлення коду і даних з простору коду в простір системи, Bus Matrix – Матриця шин, Code interface – Інтерфейс команд, SRAM & peripheral I/F – Інтерфейс SRAM і периферійних пристроїв

Сімейство Cortex також підтримує установку і скидання бітів в межах 1 Мбайта пам'яті методом, що називається "сегментація бітів", за допомогою якого здійснюється ефективний доступ до

регістрів периферійних пристроїв і прапорців, розміщених в SRAM, без залучення повноцінних булевих операцій.

Одним із ключових компонентів Cortex-M3 є контролер вкладених векторних переривань (NVIC). NVIC має стандартизованную структуру переривань для усіх мікроконтролерів на основі Cortex і відрізняється неперевершеними можливостями обробки переривань. NVIC містить вектор переривань для 240 джерел, причому кожному джерелу переривання можна індивідуально призначити пріоритет. Контролер NVIC розроблявся для дуже швидкої обробки переривань. З моменту отримання переривання до початку виконання першої команди обробника переривання витрачається тільки 12 циклів тактового сигналу. Це досягається частково за рахунок автоматичного розміщення інформації в стеку, здійснюваного усередині ЦПП мікроконтролера. У разі, коли переривання йдуть упритул один за одним, NVIC використовує метод «постановки в чергу», що дозволяє здійснювати обробку кожного з подальших переривань всього за шість циклів тактового сигналу. На етапі звернення до стека, переривання з вищим пріоритетом може отримати перевагу перед перериванням з низьким пріоритетом, що не спричиняє витрат додаткових циклів ЦПП. Логіка переривання жорстко зв'язана з режимами пониженого енергоспоживання. ЦПП може автоматично переходити в режим низького енергоспоживання при виході з переривання. Тобто ядро залишається в "сплячому" режимі до виникнення наступної виняткової ситуації.

Попри те, що ядро Cortex-M3 розроблене для бюджетних рішень, воно 32-бітове і підтримує два режими роботи: Потік і Обробник, які можуть конфігуруватися зі своїми власними стеками. Усе це дозволяє реалізовувати пристрої із складним програмним забезпеченням і підтримкою операційних систем реального часу. Cortex містить 24-бітовий таймер з автоматичним перезавантаженням, призначений для генерації періодичних переривань ядра RTOS. На відміну від ARM7 і ARM9, працюючих з двома наборами команд (32-бітові ARM і 16-бітові Thumb), сімейство Cortex підтримує одну систему команд, ARM Thumb-2. Ця система містить і 16- і 32-бітові команди, поєднуючи в собі продуктивність 32-бітового ARM зі

щільністю коду 16-бітового Thumb. Thumb-2 – це багата система команд, розроблена для роботи з Сі/Сі++ компіляторами. Це означає, що додаток під Cortex може бути повністю написаний на Сі.

1.1.1. Багатофункціональні периферійні пристрої

На перший погляд, набір периферійних пристроїв виглядає як у звичайного невеликого мікроконтролера. До його складу входить подвійний АЦП, таймери загального застосування, I2C, SPI, CAN, USB і годинник реального часу. Проте кожен з цих периферійних пристроїв має багато особливостей. Наприклад, 12-бітовий АЦП містить вбудований температурний датчик і підтримує безліч режимів перетворення, а пристрої з подвійним АЦП можуть перебудовувати обидва АЦП відразу на роботу в одному з дев'яти режимів перетворення. Так само, кожен з чотирьох таймерів містить чотири пристрої захоплення/порівняння, і кожен таймер може об'єднуватися з іншими для утворення складних масивів таймерів. Таймер з розширеними функціями призначений для додатків управління приводами, він містить шість виходів комплементу ШІМ з програмованим «мертвим часом» і вхідною лінією екстреного відключення, яка переводить виводи ШІМ у передвизначений безпечний стан. Модуль SPI містить апаратний генератор надмірного циклічного коду (CRC) для 8- і 16-бітових слів, що використовується при підключенні до SD і MMC карт пам'яті.

STM32 містить модуль прямого доступу до пам'яті (DMA) на сім каналів, що взагалі незвично для невеликих мікроконтролерів. Кожен канал може бути використаний для передачі 8/16 або 32-бітових даних в і з будь-якого регістра периферійних пристроїв. Кожен із периферійних пристроїв, залежно від завдань, може бути або відправником, або одержувачем даних від DMA контролера. Вбудовані арбітр шини і матриця шин мінімізують арбітраж між ЦПП і DMA. Усе це свідчить про гнучкість DMA, про те, що він простий у використанні і дійсно автоматизує передачу даних в межах мікроконтролера.

Якщо спробувати охарактеризувати STM32 двома словами, то це малоспоживаючий, але водночас високопродуктивний мікроконтролер. Він може працювати від джерела напруги 2 В з

тактовою частотою ядра 72 Мгц з увімкненими усіма периферійними пристроями і при цьому споживати тільки 36 мА. У режимі пониженого енергоспоживання STM32 споживає 2 мкА. Вбудований RC-осцилятор на 8 Мгц дозволяє швидко виходити з режимів пониженого енергоспоживання, доки зовнішній осцилятор стабілізується. Швидкий вхід і вихід з режимів пониженого споживання скорочує загальну енерговитратність системи.

1.1.2. Надійність

Багато сучасних пристроїв, окрім забезпечення високої продуктивності і функціональності, повинні задовольняти особливим вимогам забезпечення безпеки. STM32 має кілька апаратних особливостей для підтримки цілісності системи. До їх числа входить детектор падіння напруги живлення, система безпеки системи синхронізації і два окремі сторожові таймери. Перший сторожовий таймер віконного типу. Він повинен оновлюватися в певний часовий інтервал. Якщо оновлення станеться раніше або пізніше, сторожовий таймер спрацює, тобто згенерує переривання. Другий сторожовий таймер незалежний, має власний зовнішній осцилятор, окремий від основної системи синхронізації. Система синхронізації підтримує функції детектування несправностей основного зовнішнього осцилятора, у разі яких відбувається перемикання на вбудований 8 Мгц RC-осцилятор.

1.1.3. Безпека

Однією з вимог до сучасних пристроїв є забезпечення збереження програмного коду від несанкціонованого доступу. Для Flash пам'яті STM32 може бути встановлений захист від читання через налагоджувальний порт.

Коли захист від читання включений, Flash пам'ять також захищена від запису, щоб запобігти можливості розміщення некоректного коду в таблиці векторів переривань. Мікроконтролери STM32 також містять годинник реального часу і невелику область SRAM із живленням від батареї. Вміст цієї області автоматично стирається по перериванню від модуля запобігання втручанню в пристрій.

1.2. Огляд Cortex

Процесори Cortex – це нове покоління вбудовуваних ядер ARM. Якоюсь мірою вони відрізняються від попередніх ЦПП ARM в тому, що це вже повне процесорне ядро, що складається з ЦПП Cortex і набору системних периферійних пристроїв, що оточує його, що реалізовує в собі серце вбудовуваної системи. Оскільки існує велика кількість типів вбудовуваних систем, доступні процесори Cortex для різних профілів додатків. На профіль вказує буква в назві. На сьогоднішній момент існують такі три профілі:

Cortex-A, процесори для додатків із складними операційними системами (ОС) і алгоритмами користувача. Підтримує системи команд ARM, Thumb і Thumb-2.

Cortex-R, профіль для систем реального часу. Підтримує системи команд ARM, Thumb і Thumb-2.

Cortex-M, профіль для мікроконтролерів, оптимізованих за ціною. Підтримує тільки систему команд Thumb-2.

Число у кінці назви Cortex означає відповідний рівень продуктивності: від 0 (мінімального) до 8 (максимального). На сьогодні рівень продуктивності 4 є найвищим рівнем, доступним для профілю мікроконтролерів.

1.2.1. Ревізія архітектури ARM

Компанія ARM маркує кожний зі своїх процесорів відповідно до ревізії архітектури (ARMV6, ARMV7 і так далі). Cortex-M3 має ревізію архітектури ARMV7M. Документація для Cortex-M3 представлена "Технічним довідковим керівництвом Cortex-M3" і "Довідковим керівництвом архітектури ARMV7M". Обидва документи доступні на сайті компанії ARM: www.arm.com.

1.3. ЦПП Corte.

Серцем процесорів Cortex є 32-бітове RISC ЦПП. Ця ЦПП має спрощену версію програмної моделі ARM7/9, але розширену систему команд із доброю підтримкою цілочислової арифметики, кращою бітовою маніпуляцією і продуктивністю з "жорсткішим" реальним часом.

1.3.1. Системна архітектура STM32

Мікроконтролери STM32 побудовані на базі процесорів Cortex-M3. STM32 складається з ядра Cortex, яке з'єднується з

Flash пам'яттю через спеціальну Шину Команд. Шина Даних і Системна Шина Cortex з'єднуються з матрицею високошвидкісних шин ARM Advanced High Speed Busses (AHB).

Внутрішня SRAM підключається безпосередньо до матриці шин AHB, так само як і модуль прямого доступу до пам'яті (DMA). Периферійні пристрої розташовуються на двох шинах ARM Advanced Peripheral Bussess (APB). Кожна з APB шин замикається на матрицю шин AHB. Матриця шин тактується з тією ж частотою, що і ядро Cortex. Проте шини AHB мають окремі дільники частоти і можуть тактуватися з пониженою частотою для зниження енергоспоживання. Зауважимо, що APB2 може працювати з частотою 72 Мгц, тоді як APB1 з частотою 36 Мгц. ЦПП Cortex і DMA можуть бути "хазяїном" шини. Завдяки паралельним принципам організації матриці шин, арбітраж потрібний тільки у разі одночасного доступу до SRAM, шини APB1 або APB2.

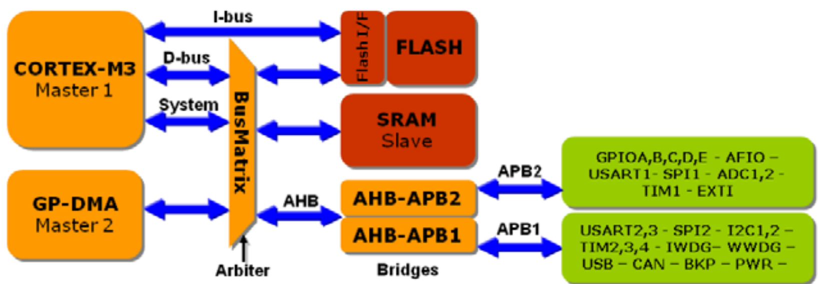


Рис. 1.2. Структура вбудованих шин: Master 1 – Ведучий 1; Master 2 – Ведучий 2; GP-DMA – DMA загального використання; BusMatrix – Матриця Шин; Flash I/F – Інтерфейс Flash; Slave – Ведений; Arbiter – Арбітр; Bridges – Мости

Структура вбудованих шин забезпечує окрему шину для команд і матрицю шин для організації доступу ядра Cortex і модуля DMA до ресурсів мікроконтролера.

1.3.2. Конвеєр

ЦПП Cortex здатне виконувати велику частину команд за один цикл. Як в ARM7 і ARM9, це досягається за допомогою триступінчатого конвеєра. Проте Cortex-M3 підтримує

прогнозування розгалужень для мінімізації кількості скидань конвеєра.

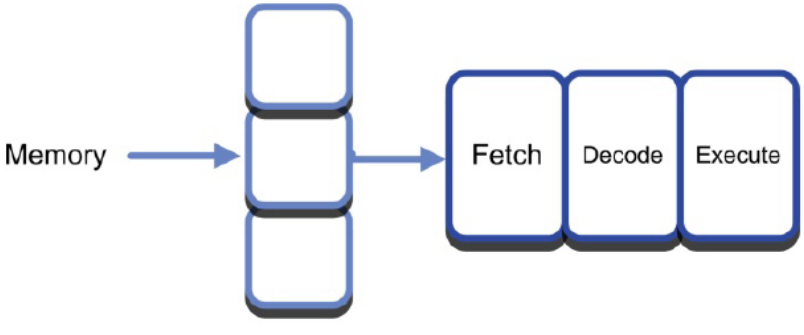


Рис. 1.3. Конвеєр виконання команд: Memory – Пам'ять; 3 Stage Prefetch – Трирівнева передвибірка; Fetch – Вибірка; Decode – Декодування; Execute – Виконання

Поки перша команда виконується, друга декодується, а третя витягається з пам'яті. Це працює дуже добре при лінійному коді, але коли зустрічається розгалуження, конвеєр скидається і завантажується наново перед тим, як програмний код продовжить виконуватися. У ARM7 і ARM9 розгалуження дуже витратні з точки зору продуктивності. У ЦПП Cortex триступінчатий конвеєр доповнений функцією прогнозування розгалужень. Це означає, що коли зустрічається команда умовного переходу, здійснюється витягання наступних передбачуваних команд. Отже, обидва варіанти умовного переходу доступні для виконання, що не спричиняє зниження продуктивності.

Гірший випадок – непрямий перехід, при якому передбачуваний перехід не може бути визначений і єдиним виходом з ситуації є скидання конвеєра. Оскільки саме конвеєр визначає повну продуктивність ЦПП Cortex, додаткові міркування відносно коду застосування були б зайвими.

1.3.3. Програмна модель

ЦПП Cortex – це RISC процесор з архітектурою load-store. Перед виконанням команд обробки даних, операнди мають бути

завантажені в центральний регістровий файл. Операції з даними повинні виконуватися в цих регістрах і результати зберігатися в пам'яті. Отже, уся програмна діяльність зосереджується на регістровому файлі ЦПП. Цей регістровий файл складається з шістнадцяти 32-бітових регістрів. Регістри R0…R12 є простими регістрами, які можуть бути використані для зберігання змінних. Регістри R13…R15 виконують спеціалізовані функції усередині ЦПП Cortex. Регістр R13 використовується як покажчик стека. Це згрупований регістр, що дозволяє ЦПП Cortex мати окремі стеки для двох режимів роботи. Зазвичай це використовується RTOS, яка виконує свій "системний" код в захищеному режимі. У ЦПП Cortex ці два стеки називаються основним стеком і робочим стеком. Наступний регістр R14 називається регістром зв'язку. Він використовується для зберігання адреси повернення при переході в процедуру. Це дозволяє ЦПП Cortex здійснювати швидкий вхід і вихід з процедури. Якщо програмний код містить декілька рівнів вкладених підпрограм, компілятор автоматично зберігатиме значення R14 в стеку при переході на наступний рівень. Останній регістр R15 є лічильником команд (PC); оскільки він входить до складу регістрового файлу, значення R15 можна зчитувати і змінювати, як у випадку з будь-яким іншим регістром.

Рис. 1.4. Центральний регістровий файл

Отже, Cortex-M3 містить регістровий файл ЦПП з 16-ти 32-бітових регістрів. Як в ARM7/9, регістр R13 є покажчиком стека. R14 – регістр зв'язку і R15 – лічильник команд. Регістр R13 згрупований, що дозволяє Cortex-M3 працювати з двома стеками: робочим і основним.

1.3.4. Регістр xPSR

На доповнення до регістрового файлу існує окремий регістр під назвою Program Status Register. Він не входить до складу головного регістрового файлу і доступний тільки через дві спеціальні команди. Значення у бітових областях регістра xPSR впливають на роботу ЦПП Cortex. Цей регістр також називають Application, Execution and Interrupt Status Register.

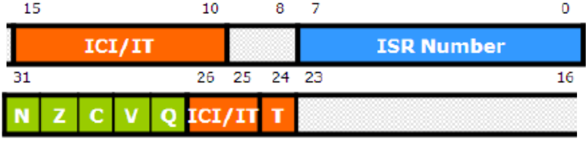


Рис. 1.5. Області Program Status Register, що впливають на виконання команд

Регістр xPSR може бути доступний також через три спеціальні псевдоімені, що дозволяють звертатися до певних областей xPSR. П'ять верхніх бітів є прапорами коду умови і доступні під псевдоім'ям Application Program Status Register. Перші чотири прапори коду умови, N, Z, C, V (Negative, Zero, Carry і Overflow) встановлюються і скидаються залежно від результату виконання команди. Біт Q використовується DSP-командами, щоб показати, що змінна досягла свого максимального або мінімального значення. Як і в 32-бітовій системі команд ARM, команди Thumb-2 виконуються тільки, якщо код умови команди відповідає стану прапорів регістра Application Program Status Register. Якщо відповідності немає, команда проходить через конвеєр як порожня команд NOP. Це забезпечує "плавну течію" команд через конвеєр і мінімізує кількість скидань конвеєра. У ЦПП Cortex ця технологія розширена за допомогою регістра Execution Program Status Register. Це псевдоім'я бітів 26…8 регістра xPSR. Він включає три області: "If then", "команда продовжується після переривання" і "команда Thumb". Система команд Thumb-2 підтримує ефективний метод виконання блоку команд "if then". Якщо умова виконується, встановлюється значення в області "if then", яке змушує ЦПП виконувати до чотирьох наступних команд. Якщо умова не виконується, ці команди проходять через конвеєр як NOP. Таким чином, типовий рядок мовою Сі компілюватиметься, як показано нижче:

```
If (r0 == 0)
    CMP r0,#0 перевірка r0 з 0
    ITTEE EQ якщо “істина”, виконуємо наступні дві команди
Then r0 = *r1 + 2;
    LDR r0,[r1] завантажуємо вміст з пам’яті в r0
    ADD r0,#2 додаємо 2
```

Попри те, що більшість команд Thumb-2 виконуються за один цикл, деякі (такі як команди завантаження і збереження) вимагають кількох циклів.

Таким чином, для того, щоб ЦПП Cortex міг мати детермінований час реакції на переривання, такі команди мають бути такими, що перериваються.

Коли команда переривається, в області "команда продовжувана після переривання" зберігається кількість

наступних регістрів, які треба обробити згідно з командою. Відповідно, після завершення обробки переривання, багатоциклова команда може продовжувати виконання. Остання область, "команда Thumb", успадкована з попередніх ЦПП ARM. Ця облать показує до якої системи належить виконувана ЦПП команда: ARM або Thumb. У Cortex-M3 цей біт завжди встановлений в одиницю. На закінчення, область статусу переривання містить інформацію про запити переривання, які були відкладені.

1.3.5. Карта пам'яті

Процесор Cortex-M3 – це стандартне мікроконтролерне ядро, тому містить чітко фіксовану карту пам'яті. Незважаючи на множину внутрішніх шин, карта пам'яті лінійна із 4 Гбайтним адресним простором.

Cortex-M3 має фіксовану 4 Гб карту пам'яті з вказаними областями для коду, SRAM, периферійних пристроїв, зовнішньої пам'яті і пристроїв і системних регістрів Cortex. Ця карта пам'яті загальна для усіх пристроїв на базі Cortex.

Перший 1 Гбайт пам'яті розділений порівну між областями для програмного коду і для SRAM. Область для коду доступна через шину I-Code, а область SRAM через шину D-code. Попри те, що програмний код може завантажуватися і виконуватися з SRAM, команди у такому разі витягатимуться з використанням системної шини, що призводить до додаткових затримок. Найімовірніше код виконуватиметься повільніше з SRAM, ніж із вбудованої Flash пам'яті, розташованої в області для програмного коду. Наступні 0.5 Гбайт пам'яті виділено під вбудовані периферійні пристрої. Усі регістри периферійних пристроїв, що надаються виробником, будуть розташовані в цій області. Перший 1 Мбайт областей для SRAM і периферійних пристроїв є таким, що побітно адресується, що досягається за допомогою техніки, яка називається "бітова сегментація". Оскільки уся SRAM і призначені для користувача периферійні облаштування STM32 розташовані саме на цій ділянці, уся пам'ять STM32 може бути доступна у форматі слів або бітів. Наступні 2 Гбайти адресного простору виділені під зовнішню SRAM і зовнішні периферійні пристрої. Останні 0.5 Гбайт призначено для внутрішніх процесорних периферійних

пристроїв Cortex і для подальших специфічних поліпшень процесора Cortex.

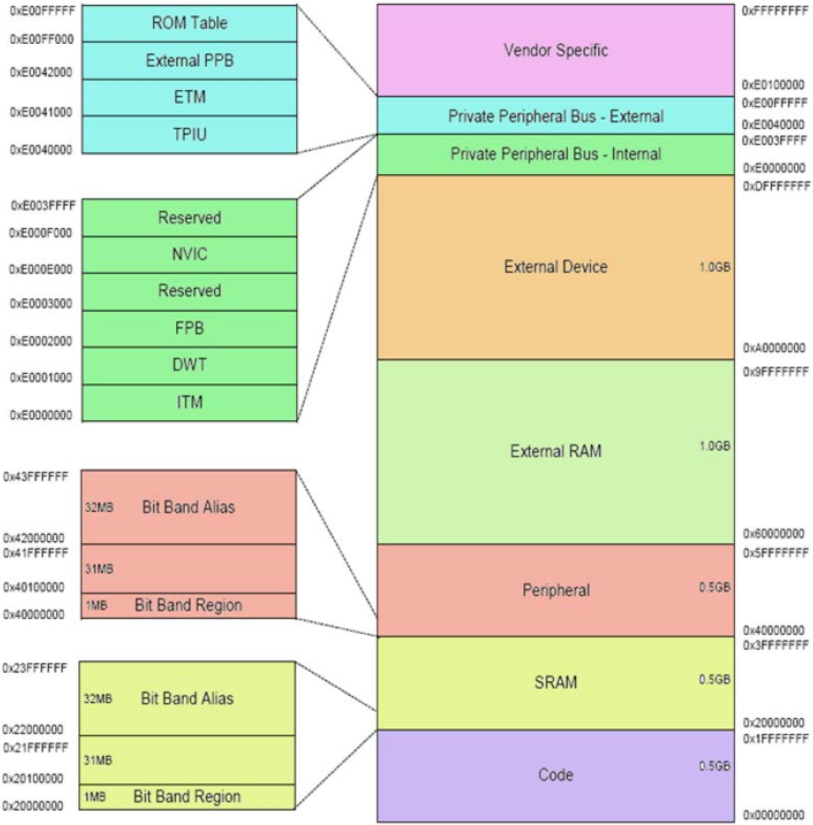


Рис. 1.6. Карта пам'яті Cortex-M3: Reserved – Зарезервовано; Bit Band Alias – Область псевдоімен бітових сегментів; Bit Band Region – Область бітових сегментів; Vendor Specific – Визначається виробником; Private Peripheral Bus - External – Приватна шина периферійних пристроїв – зовнішня; Private Peripheral Bus - Internal – Приватна шина периферійних пристроїв – внутрішня; External Device – Зовнішні периферійні пристрої; External RAM – Зовнішня RAM; Peripheral – Периферійні пристрої; Code – Програмний код

Усі регістри процесора Cortex мають фіксоване місце розташування в усіх мікроконтролерах на базі Cortex. Це

дозволяє легко переносити код між різними варіантами STM32 і навіть на Cortex-мікроконтролери інших виробників.

1.3.6. Розподіл пам'яті

Попри те, що STM32 містить безліч внутрішніх шин, програмістові відображається лінійний 4 Гбайтний адресний простір. Оскільки STM32 – це мікроконтролер на базі Cortex, карта пам'яті відповідає стандартному розподілу пам'яті Cortex. Отже, пам'ять програм розпочинається з адреси 0x00000000. Вбудована SRAM починається з 0x20000000 і уся вона розташовується в області з бітовою сегментацією. Пам'ять призначених для користувача периферійних пристроїв розпочинається з адреси 0x40000000 і також розташована в області з бітовою сегментацією. Регістри Cortex розташовані, починаючи з адреси 0xE0000000.

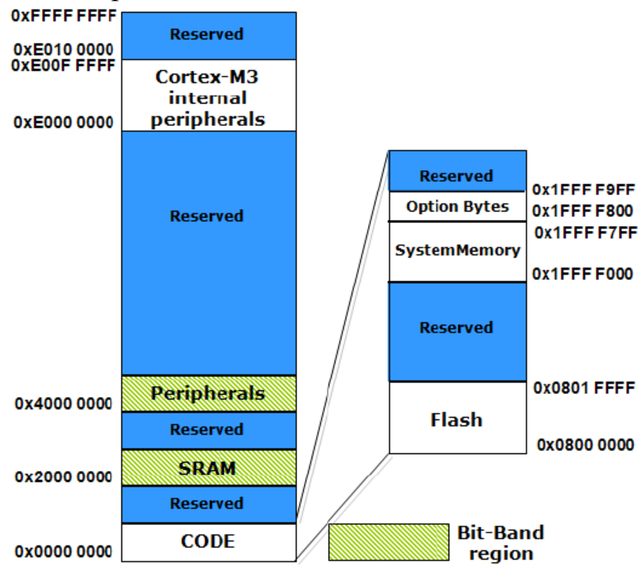


Рис. 1.7. Розподіл пам’яті: Reserved – Зарезервовано; Cortex-M3 internal peripherals – Внутрішні периферійні пристрої Cortex; Peripherals – Периферійні пристрої; CODE – програмний код; Option Bytes – Опциональні байти; System Memory – Системна пам'ять; Bit-Band region – Область з бітовою сегментацією

Таблиця 1.1

Режим завантаження

Виводи мікроконтролера вибору режиму завантаження		**Режим завантаження**	**Пояснення**
BOOT1	**BOOT0**		
x	0	User Flash (Flash пам’ять користувача)	Користувацька Flash вибрана як завантажувальна область
0	1	System memory (системна пам'ять)	Системна пам'ять вибрана як завантажувальна область
1	1	Embedded SRAM (вбудована SRAM)	Вбудована SRAM вибрана як завантажувальна область

Карта пам'яті STM32 відповідає стандартам Cortex. Перші 2 Кбайт пам'яті адресуються в Flash, Системну Пам'ять або SRAM залежно від стану виводів вибору режиму завантаження.

Область FLASH пам'яті складається з трьох секцій. Перша – це Призначена для користувача Flash пам'ять, розпочинається з адреси 0x00000000. Друга – Системна Пам'ять, що також називається великим інформаційним блоком. У ці 4 КБайт Flash пам'яті на етапі виробництва мікроконтролера запрограмований завантажувач. Третя секція, що розпочинається з адреси 0x1FFFF800, називається малим інформаційним блоком, містить групу опційних байтів, які дозволяють здійснювати системні налаштування STM32. Завантажувач розроблений з метою завантаження коду через USART1 в Призначену для користувача FLASH пам'ять. Для переведення STM32 в режим завантажувача, зовнішні виводи BOOT0 і BOOT1 мають бути переведені відповідно в низький і високий стани. У такому стані виводів блок системної пам'яті відображається за адресою 0x00000000. Після скидання, замість виконання програми користувача з Призначеної для користувача Flash, STM32 почне виконувати код завантажувача. Застосунок завантажувача для ПК доступний для скачування на сайті компанії ST. Цей застосунок використовується для обміну інформацією із завантажувачем, стирання і перепрошивки Призначеної для користувача Flash пам'яті. Застосунок для ПК також доступний у вигляді DLL, що дозволяє написати свій власний завантажувач для оновлення областей пам'яті і для програмування

мікроконтролерів на етапі серійного виробництва пристроїв. Виводи вибору режиму завантаження дозволяють також відображати вбудовану SRAM за адресою 0x00000000 замість Призначеної для користувача Flash. Це дає можливість на етапі розробки завантажувати і виконувати програмний код безпосередньо з SRAM. При цьому прискорюється процес завантаження програми і зменшується кількість циклів запису/стирання FLASH пам'яті, що збільшує термін її служби.

1.3.7. Бітова сегментація

Технологія бітової сегментації дозволяє здійснювати атомарні бітові маніпуляції.

Попередні версії ARM7 і ARM9 могли здійснювати бітові маніпуляції в SRAM і областях пам'яті для периферійних пристроїв тільки з використанням операцій І та АБО. При цьому вимагалося виконувати команду "ЧИТАННЯ-ЗМІНА-ЗАПИС", що дуже витратно з точки зору кількості циклів на встановлення і скидання бітів і кількості команд, необхідних для кожної маніпуляції з бітами.

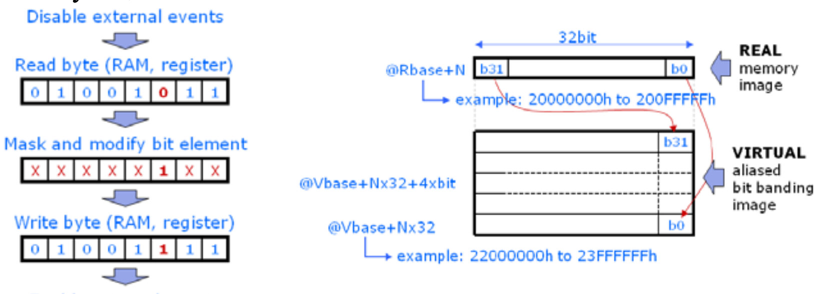


Рис. 1.8. Технологія бітової сегментації: Disable external events – Заборона зовнішніх подій; Read byte (RAM, register) – Читання байта (ОЗП, регістр); Mask and modify bit element – Маскування і зміна бітів; Write byte (RAM, register) – Запис байта (ОЗП, регістр); Enable external events – Дозвіл зовнішніх подій; REAL memory image – РЕАЛЬНА пам'ять; VIRTUAL aliased bit banding image – ВІРТУАЛЬНІ псевдоімена бітових сегментів; example – приклад

Для подолання цього обмеження, можна б було ввести спеціальні команди установки і скидання бітів, або повний Булевий процесор, але це б привело до збільшення розміру і

складності ЦПП. Впровадження технології бітової сегментації в областях пам'яті SRAM і периферійних пристроїв дозволяє здійснювати бітову маніпуляцію без необхідності в додаткових командах. Область з побітовою адресацією карти пам'яті Cortex складається з області бітових сегментів (розміром до 1 МБ реальної пам'яті і регістрів периферійних пристроїв) і області псевдоімен бітових сегментів, яка займає до 32 МБ карти пам'яті. При бітовій сегментації кожному біта в області бітових сегментів відповідає адреса слова з області псевдоімен. Таким чином, встановлюючи або скидаючи значення по адресах слів псевдоімен, можна встановлювати і скидати біти в реальній пам'яті.

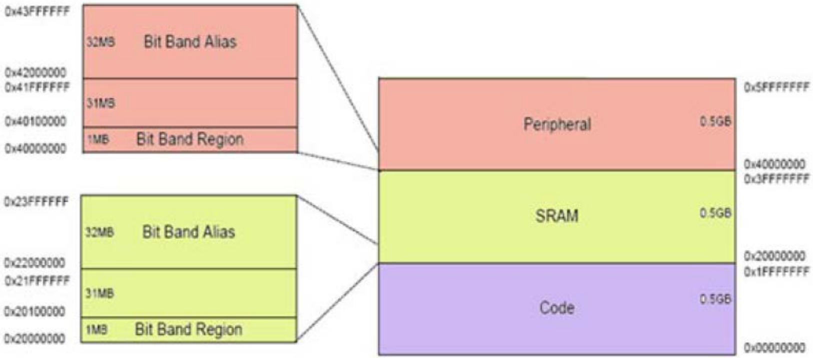


Рис. 1.9. Області з побітовою адресацією: Bit Banding Alias – Область псевдоімен Бітових Сегментів; Bit Band Region –Область Бітових Сегментів; Periferal – Периферійні пристрої; Code – Програмний Код

Бітова сегментація працює в межах першого 1 Мбайта областей SRAM і Периферійні пристрої. У цих областях укладаються всі ресурси STM32.

Така технологія дозволяє нам здійснювати бітові маніпуляції без необхідності в спеціальних командах і збільшенні розмірів ядра Cortex. На практиці, необхідно обчислювати адресу псевдоімені бітового сегмента для заданого елемента пам'яті області SRAM або периферійних пристроїв. Формула для обчислення адреси псевдоімені наведена нижче:

Адреса в області псевдоімен = Початкова адреса області псавдоимен + зміщення слова

де

Зміщення слова = Зміщення у байтах від початку бітових сегментів x 0x20 + номер біта x 4

Усе це набагато простіше, ніж здається на перший погляд. Для прикладу розглянемо, як здійснюється запис у вихідний регістр цих портів введення/виведення загального призначення (GPIO) для встановлення і скидання окремих ліній введення/виведення. Фізична адреса вихідного регістра PORT B 0x40010C0C. У цьому прикладі встановлюватимемо і скидатимемо восьмий біт у слові, використовуючи наведену вище формулу.

Адреса слова = 0x40010C0C

Базова адреса області бітових сегментів периферійних пристроїв = 0x40000000

Базова адреса області псевдоімен бітових сегментів периферійних пристроїв = 0x42000000

Байтове зміщення від початку бітових сегментів = 0x40010C0C - 0x40000000 = 10C0C

Зміщення слова = (0x10C0C x 0x20) + (8x4) = 0x2181A0

Адреса в області доступу до бітів = 0x42000000 + 0x2181A0 = 0x422181A0

Тепер ми можемо створити покажчик на цю адресу, використовучи такий рядок мови Сі:

```
#define PortBbit8(*((volatile unsigned long*) 0x422181A0 ))
```

Цей покажчик потім може використовуватися для встановлення і скидання біта порту введення/виведення:

```
PB8 = 1; //встановити біт
```

Унаслідок цього генеруються такі асемблерні команди:

```
MOVS r0, #0x01
LDR     r1, [pc, #104]
STR     r0, [r1, 0x00]
```

Вимкнення світлодіода:

```
PB = 0; //скинути біт
```

Генеруються такі асемблерні команди:

```
MOVS r0, #0x00
LDR     r1, [pc, #88]
STR     r0, [r1, 0x00]
```

Кожна з операцій встановлення і скидання займає три 16-бітові команди, і на STM32, працюючому з частотою 72 Мгц, ці команди виконаються за 80 нс. Будь-яке слово у бітовій сегментованій області периферійних пристроїв і пам'яті SRAM

може також безпосередньо адресуватися у форматі слова, так що ми можемо здійснити те ж саме встановлення і скидання, використовуючи традиційний підхід з операціями I та АБО:

```
GPIOB -> ODR |= 0x00000100; //встановити біт
    LDR        r0, [pc, #68]
    ADDS       r0, r0, #0x08
    LDR        r0, [r0, #0x00]
    ORR        r0, r0, #0x100
    LDR        r0, [pc, #64]
    STR        r0, [r1, #0xC0C]

GPIOB -> ODR &= ! 0x00000100; //скинути біт
    LDR        r0, [pc, #40]
    ADDS       r0, r0, #0x08
    LDR        r0, [r0, #0x00]
    MOVS r0, #0x00
    LDR        r1, [pc, #40]
    STR        r0, [r1, #0xC0C]
```

Кожна з операцій встановлення і скидання вимагає групи з 16- і 32-бітових команд, що споживає як мінімум 14 байт пам'яті для кожної операції і 180 нс часу при тій же тактовій частоті. У звичайному вбудовуваному застосуванні дуже часто використовуються операції установки і скидання бітів в регістрах периферійних пристроїв, семафорів і прапорів у SRAM. І якщо взяти до уваги переваги технології бітової сегментації, то можна значно скоротити об'єм коду застосунку і час його виконання. Тим більше, в заголовних файлах STM32 вже проведені всі необхідні розрахунки для використання цієї технології.

1.4. Процесор Cortex

1.4.1. Шини

Процесор Cortex-M3 побудований по Гарвардській архітектурі пам'яті з роздільними шинами для коду і даних. Ці шини мають назви Icode і Dcode. Обидві шини мають доступ до програмного коду і даних в діапазоні адрес 0x00000000…0x1FFFFFFF. Додаткова системна шина використовується для доступу до системної керуючої області Cortex в діапазоні адрес 0x20000000…0xDFFFFFFF та 0xE0100000…0xFFFFFFFF. Вбудована система налагодження Cortex має додаткову структуру шин – Private Peripheral Bus.

1.4.2. Матриця шин

Системна шина і шина даних підключаються до зовнішнього мікроконтролера через сукупність високошвидкісних шин, організованих у вигляді матриці. Це дозволяє реалізовувати безліч паралельних каналів зв'язку між шинами Cortex і зовнішніми пристроями управління передачею даних по шині, наприклад, між DMA і SRAM. Якщо два пристрої управління передачею даних по шині (тобто ЦПП Cortex і канал DMA) намагаються отримати доступ до одного й того самого периферійного пристрою, вбудований арбітр вирішить конфлікт і надасть доступ пристрою з вищим пріоритетом. Як ми побачимо при розгляді модуля DMA, STM32 побудований так, що DMA і ЦПП Cortex працюють погоджено.

1.4.3. Системный таймер SysTick

Таймер SysTick – це 24-бітовий таймер у складі процесора Cortex-M3, що автоматично перезавантажується. Він призначений для відліку інтервалів Операційної Системи Реального Часу.

Ядро Cortex містить 24-бітовий віднімаючий лічильник, з функціями автоперезавантаження і генерації переривання після завершення лічби. Це стандартний таймер для усіх мікроконтролерів на базі Cortex. Таймер SysTick може використовуватися для відліку системних інтервалів RTOS або для генерації періодичних переривань запланованих завдань.

Таймер SysTick містить три регістри: Control & Status, Reload Value, Current Value. Джерело тактування для таймера SysTick налаштовується за допомогою регістра SysTick Control і регістра Status в області управління Cortex-M3 System. При встановленому біті CLKSOURSE таймер SysTick працюватиме на частоті ЦПП. Якщо цей біт скинутий, таймер працюватиме на частоті в 8 разів меншій від частоти ЦПП.

Поточне значення і значення перезавантаження повинні встановлюватися разом з періодом рахунку. Регістр управління і статусу містить біт ENABLE для запуску таймера і біт TICKINT для включення лінії переривань.

1.4.4. Контролер вкладених векторних переривань NVIC

Контролер Вкладених Векторних Переривань (NVIC) – це стандартний модуль ядра Cortex. Це означає, що будь-який мікроконтролер на базі Cortex матиме ідентичну структуру переривань, незалежно від виробника. Таким чином, код застосування і операційна система можуть легко переноситися з одного мікроконтролера на інший і програмістові не доведеться освоювати новий набір регістрів. NVIC розроблений забезпечувати дуже малі затримки переривання. Це досягається за допомогою самого NVIC і системи команд Thumb-2, яка дозволяє переривати багатоциклові команди. Затримка переривання також детермінована, є спеціальні можливості обробки переривань для підтримки застосувань реального часу. Контролер Вкладених Векторних Переривань підтримує вкладені переривання і в STM32 є 16 рівнів пріоритету. Структура переривань програмується повністю на Сі і не вимагає ніяких макросів Асемблера або спеціальних не-ANSI директив.

Попри те, що NVIC є стандартним модулем у складі ядра Cortex, для того, щоб кількість вентилів залишалася мінімальною, кількість ліній переривань, що надходять на вхід NVIC, конфігурується на етапі розробки мікроконтролера. NVIC містить одне немасковане переривання і до 240 ліній зовнішніх переривань, які можуть бути сполучені з призначеними для користувача периферійними пристроями. Наявні також додаткові 15 джерел переривань усередині ядра Cortex, використовувані для обробки внутрішніх виняткових ситуацій самого ядра Cortex. NVIC STM32 був синтезований з максимальною кількістю ліній маскованих переривань рівним 43.

1.4.4.1. NVIC: функціонування, виключення, вхід і вихід

При виникненні переривання від периферійного пристрою, NVIC починає переведення ЦПП Cortex в режим обробки переривання. Як тільки ЦПП Cortex входить в режим обробки переривання, він зберігає значення набору регістрів в стек. Важливе те, що це здійснює мікрокод, тобто непотрібні

додаткові команди в коді застосунку. В той час як в стеку зберігаються дані, витягається початкова адреса процедури обробки переривань. Таким чином, з моменту виникнення переривання до початку виконання першої команди процедури обробки переривання проходить тільки 12 циклів.

У стеку зберігаються значення регістра Program Status Register, лічильника команд і значення регістра зв'язку, щоб запам'ятати стан ЦПП Cortex на момент виникнення переривання. Крім того, значення регістрів R0…R3 теж зберігаються. У стандарті бінарного інтерфейсу ARM ці регістри використовуються для передачі даних, тому, зберігши їх значення, ми збільшуємо кількість регістрів, які можуть використовуватися в процедурі обробки переривань. І останнє, значення регістра R12 також зберігається; це робочий регістр внутрішніх викликів. Він використовується кожною програмою, яка виконується під час виклику функції. Наприклад, якщо ви активували перевірку стека під час компіляції, згенерований код використовуватиме регістр R12, якщо буде необхідність в регістрі ЦПП. Після завершення обробки переривання, усі дії виконуються в зворотному порядку, мікрокод відновлює інформацію зі стека, одночасно з цим витягається адреса повернення. Виконання програмного коду поновлюється через 12 циклів.

1.4.4.2. Розширений режим обробки переривань

Окрім швидкої обробки поодиноких переривань, в NVIC на етапі розробки закладалася можливість ефективної обробки декількох переривань застосунку реального часу. У NVIC реалізовано декілька методів обробки переривань від безлічі джерел з мінімальною затримкою між перериваннями, які гарантують, що переривання з вищим пріоритетом буде оброблено в першу чергу.

1.4.4.3. Пріоритетне обслуговування переривань

NVIC розроблений так, що переривання з великим пріоритетом буде оброблено без черги, навіть якщо у цей момент уже обробляється переривання з меншим пріоритетом. При цьому обробка переривання з меншим пріоритетом зупиняється, новий стековий фрейм зберігається за стандартні 12 циклів, після чого запускається обробник переривання з

великим пріоритетом. Після завершення обробки переривання з великим пріоритетом, дані зі стека автоматично виштовхуються, і поновлюється обробка переривання з меншим пріоритетом.

1.4.4.4. Конфігурація і використання NVIC

Для використання NVIC треба зробити три речі. Перше, конфігурувати таблицю векторів для джерел переривань, які ми хочемо використовувати. Друге, встановити пріоритети і дозволити переривання, конфігурувавши регістри NVIC. І третє, налаштувати периферійні пристрої і дозволити їх переривання.

1.4.4.5. Таблиця векторів переривань

Таблиця векторів переривань Cortex знаходиться внизу адресного простору. Проте замість того, щоб розпочинатися з нульової адреси, першою адресою таблиці векторів переривання є 0x00000004. Перші чотири байти використовуються для зберігання початкової адреси покажчика стека.

Таблиця винятків Cortex містить початкові адреси або адреси процедур обробки переривань, які завантажуються в лічильник команд при виникненні винятків.

Кожен вектор переривання має розмір чотири байти і містить початкову адресу тієї процедури обробки переривання, з якою пов'язано це переривання. Перші 15 векторів призначено для винятків, що відбуваються усередині самого ядра Cortex. До їх числа входить вектор скидання, немасковане переривання, помилка управління, виключення налагоджувальної системи, а також переривання від таймера SysTick. Система команд Thumb-2 також містить команди виклику системних сервісів, при виконанні яких генеруються переривання. Переривання від призначених для користувача периферійних пристроїв розпочинаються з 16-го вектору і пов'язані з периферійними пристроями так, як визначив виробник.

У програмному коді таблиця векторів переривань, зазвичай на початку, заповнюється адресами процедур обробки переривань.

Таблиця 1.2

Вектори переривань

№	Тип винятку	Пріо-ритет	Тип пріоритету	Опис
1	Reset	-3 (найви-щий)	фіксований	Скидання
2	NMI	-2	фіксований	Немасковане переривання
3	Hard Fault	-1	фіксований	Помилка за умовчанням, якщо інші пункти незастосовувані
4	MemManage Fault	0	задається	Порушення МПП або звернення до забороненої області
5	Bus Fa ult	1	задається	Помилка, якщо інтерфейс AHB прийняв помилку
6	Usage Fault	2	задається	Виняток через програмну помилку
7…10	Зарезервовано	недо-ступно	недоступно	
11	SVCall	3	задається	Виклик Системного Сервісу
12	Dedug Monitor	4	задається	Точки останову, точки перегляду, зовнішнє налагодження
13	Зарезервовано	недо-ступно	недоступно	
14	PendSV	5	задається	«Підвішуваний» запит для Системного Пристрою
15	SYSTICK	6	задається	Системний таймер System Tick
16	Переривання #0	7	задається	Зовнішнє переривання #0
…	…	…	…	…
256	Переривання #240	247	задається	Зовнішнє переривання #240

```
    AREA      RESET, DATA, READONLY
    EXPORT       __Vectors
__Vectors
    DCD   __initial_sp               ;Вершина стеку
    DCD   Reset_Handler ;Обробник скидання
    DCD   NMI_Handler                ;Обробник NMI (немаскованого
                                      переривання)
    DCD   HardFault_Handler          ;Обробник аварії типу HardFault
                                      (апаратної помилки)
    DCD   MemManage_Handler          ;Обробник аварії блоку захисту
                                      пам'яті (помилки МПП)
    DCD   BusFault_Handler           ;Обробник аварії типу BusFault
                                      (помилки шини)
```

```
DCD   UsageFault_Handler    ;Обробник аварії типу UsageFault
                            (помилки використання)
DCD   0                     ;Зарезервовано
DCD   0                     ;Зарезервовано
DCD   0                     ;Зарезервовано
DCD   0                     ;Зарезервовано
DCD   SVC_Handler           ;Обробник SVCall (програмно-
                            згенерованого переривання)
DCD   DebugMon_Handler      ;Обробник монітору
                            відладжувальної системи
DCD   0                     ;Зарезервовано
DCD   PendSV_Handler        ;Обробник PendSV
DCD   SysTick_Handler       ;Обробник SysTick (переривання
                            системного таймера)
```

У випадку з таймером SysTick ми можемо створити процедуру обробки переривання, оголосивши Сі-функцію з ідентичним символьним ім'ям:

```
void SysTick_Handler (void) {
....
}
```

Тепер, із заданою таблицею векторів переривань і з оголошеним прототипом функції обробки переривань, ми можемо конфігурувати NVIC на обробку переривання від таймера SysTick. У цілому, нам треба виконати дві речі: встановити пріоритет переривання і увімкнути джерело переривання. Регістри NVIC розташовані в системній області управління.

1.5. Система тактування та синхронізації

STM32 містить складну систему синхронізації з двома зовнішніми і двома внутрішніми осциляторами, а також системою ФАПЧ. Зовнішній Високочастотний Осцилятор може контролюватися системою безпеки системи синхронізації.

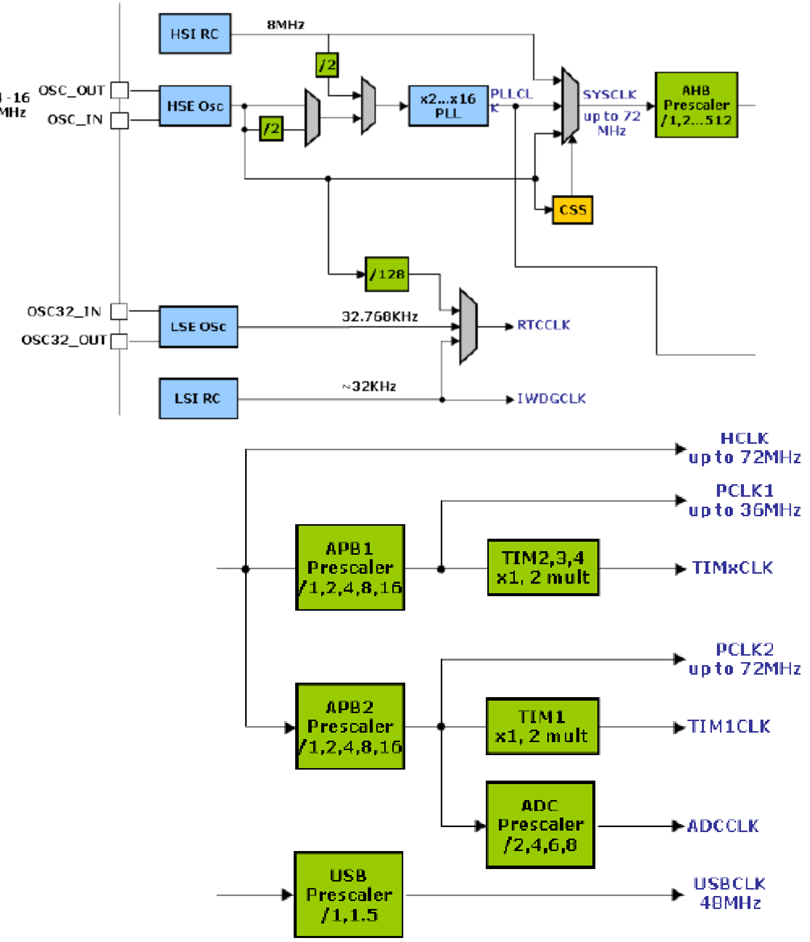


Рис. 1.10. Блок тактування та синхронізації: HSI – внутрішній високочастотний осцилятор; LSI – внутрішній низькочастотний осцилятор; HSE – зовнішній високочастотний осцилятор; LSE – зовнішній низькочастотний осцилятор; PLL – ФАПЧ; AHB (Advanced Highspeed Bus) – Розширена високошвидкісна шина; APB (Advanced Peripheral Bus) – Розширена периферійна шина; Prescaler – переддільник; TIM – таймер; ADC – АЦП; USB – послідовна шина

Процесор Cortex може тактуватися від внутрішнього або зовнішнього високочастотного осцилятора (HSE) або від внутрішньої ФАПЧ (PLL). ФАПЧ, у свою чергу, тактується від внутрішнього, або від зовнішнього високочастотного осцилятора. Тобто STM32 може працювати з частотою 72 Мгц без використання зовнішніх осциляторів. Мінус в тому, що вбудовані осцилятори не такі точні і стабільні, як зовнішні. Тому для організації послідовних інтерфейсів або реалізації точних часових функцій необхідно використовувати зовнішній осцилятор. Незалежно від того, який осцилятор використовується, для отримання 72 Мгц для ядра Cortex треба задіювати ФАПЧ. Усі регістри конфігурації ФАПЧ і шин розташовані в групі регістрів Reset and Clock Control (RCC).

На додаток до двох зовнішніх осциляторів, STM32 містить два внутрішніх RC-осцилятори. Після скидання, первинним джерелом тактового сигналу для ядра Cortex є Високочастотний Внутрішній Осцилятор (HSI) із номінальною частотою 8 Мгц. Другий внутрішній осцилятор є Низькочастотним Осцилятором (LSI), що працює на частоті 32.768 кГц. Цей осцилятор призначений для годинника реального часу і сторожових таймерів.

1.5.1. ФАПЧ

Після скидання ЦПП STM32 тактується від осцилятора HSI. У цей момент зовнішній осцилятор вимкнений. Перше, що треба зробити для запуску STM32 на повній швидкості, полягає у включенні зовнішнього осцилятора й очікуванні, доки він стабілізується.

1.6. Література до розділу

1. Trevor Martin, Sarah Latchford The insiderr’s guide to the STM32 ARM based microcontroller. An engineer’s introdaction to the STM32 series: University Of Warwick Science Park, United Kingdom, 2008.– 96p..– Режим доступу: https://www.hitex.com/fileadmin/documents/tools/dev_tools/dt_protected/insiders-guides/stm32/isg-stm32-v18d-scr.pdf

2. STM32F334x4 STM32F334x6 STM32F334x8 Arm®Cortex®-M4 32b MCU+FPU,up to 64KB Flash,16KB SRAM, 2 ADCs,3 DACs,3 comp.,op-amp, 217ps 10-ch (HRTIM1).– Ресурс: https://www.st.com/resource/en/datasheet/stm32f334k8.pdf
3. RM0364 Reference manual STM32F334xx advanced Arm®-based 32-bit MCUs.– Ресурс: https://www.st.com/resource/en/reference_manual/rm0364-stm32f334xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf
4. https://www.st.com/resource/en/datasheet/stm32f334r8.pdf
5. https://www.st.com/resource/en/reference_manual/rm0364-stm32f334xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

РОЗДІЛ 2. Стандарти програмного інтерфейсу мікроконтролерів Cortex та бібліотеки для роботи з STM32

2.1. Бібліотеки взаємодії програм з ARM Cortex

ARM - архітектура одна з найбільш популярних. Фірма-розробник ARM Holding самостійно процесорів не випускає. Вона розробляє архітектуру і систему команд та ліцензує її іншим фірмам. Виробники можуть виготовляти процесори або мікроконтролери (System on Chip), що базуються на ядрі ARM, доповнених різними периферійними компонентами, пам'яттю, сопроцесорами тощо, або розробляти власні архітектури на базі системи команд та архітектурних рішень ARM. Тому під абревіатурою ARM можуть матися на увазі різні процесори. У всіх пристроях сімейства багато спільного, але завжди є відмінності. Тому варто розуміти, про який саме ARM іде мова.

Існує багато "версій" ядра ARM, які грубо можна поділити на три групи:

- ARM Cortex-A – "звичайні" процесори, їх, наприклад, можна зустріти в ноутбуках і планшетах;
- ARM Cortex-R – процесори для систем реального часу;
- ARM Cortex-M – мікроконтролери на базі архітектури ARM.

Надалі мова йтиме лише про "мікроконтролерні" варіанти ARM Cortex-M, які виділяються великою потужністю, однак вони поступаються серії A. Крім того, позаяк для мікроконтролерів критичний розмір програм, "Кортекси" підтримують лише "урізані" системи команд -- thumb (виключно 16-бітні команди) і thumb-2 (як 16-бітні, так і 32-бітні). Досягнуто це урізання та "запаковування" у 16 біт зменшенням кількості дозволених комбінацій операндів, перетворення частини операндів на неявні тощо.

Програмування будь-якого мікроконтролера – це робота з регістрами. Через регістри відбувається управління всіма ресурсами контролера. Через регістри ми працюємо з

комунікаційними інтерфейсами і виводами, тобто зв'язуємо мікроконтролер із зовнішнім світом.

На сьогодні наявні три варіанти написання програм для STM32 з використанням нижченаведених бібліотек.

- **CMSIS** (Cortex Microcontroller Software Interface Standard, стандарт програмного інтерфейсу мікроконтролера) – це стандартна бібліотека для Cortex-M, основа її розроблена ARM Holdings, дописується виробниками мікроконтролерів під їх периферію. У ній визначено всі регістри (тому немає потреби вручну задавати їх адреси в пам'яті), різноманітні константи, бітмаски і т.д. За допомогою цієї бібліотеки ми можемо писати програми, але тільки з прямим записом в регістри. Це найпрофесійніший спосіб написання програм, і до речі, найправильніший, але складніший, оскільки вимагає досконалого володіння інформацією з документації на мікроконтролер.
- **SPL** (Standard Peripherals Library, стандартна бібліотека периферійних блоків) — це спроба ST Electronics випустити одну загальну бібліотеку для об'єднання усіх своїх процесорів, написана мовою Сі. Щоб було простіше переносити код і тому подібне. Містить функції, структури і макроси для полегшення роботи з периферією мікроконтролера. Працювати і писати код дещо простіше, але все одно вимагає від програміста знання цієї бібліотеки і «ручної» ініціалізації.
- **HAL** (Hardware Acess Level, рівень абстрагування устаткування) – це друга спроба ST Electronics випустити єдину бібліотеку для розробки. Насправді це те ж саме, що й стандартна бібліотека SPL. Бібліотека HAL є основним методом розробки, який нині просуває ST. Разом з нею вийшла і програма STM32 CubeMX для генерування шаблонного програмного коду з налаштуваннями (ініціалізації) мікроконтролера.

Створення програмного забезпечення – головний чинник витрат в індустрії вбудовуваних продуктів. За рахунок стандартизації програмних інтерфейсів для усіх продуктів постачальників мікросхем Cortex-M (особливо при створенні

нових проєктів або перенесенні існуючого програмного забезпечення на нові пристрої) можна значно понизити витрати.

Отже, зазвичай перед початком роботи з STM32 потрібно спочатку обрати метод розробки, який використовуватиметься. Вибір методу впливає на архітектуру програмування:

- Використовуючи пряме звернення до регістрів CMSIS. У цьому випадку код програми буде коротший, програма працювати швидше. Але розробка застосунку значно ускладниться. У STM32 дуже багато регістрів, багато режимів роботи, складна периферія.
- При використанні бібліотеки HAL усе навпаки. Код довший, програма повільніша, але розробляти програму простіше.

Безпосереднє управління регістрами ближче до ядра, але важке в застосуванні у STM32, тому що кількість регістрів STM32 вдесятеро більша, ніж у однокристальних восьмибітних мікроконтролерів, і всіх їх повністю запам'ятати складно. Під час розробки програми часто необхідно користуватися документацією на мікроконтролер.

Як уже згадувалося вище, STM32 має багато регістрів, що викликає труднощі при розробці програми, тому компанія ST написала файл стандартної бібліотеки SPL для кожного чіпа, який є файлом проєкту. У ці файли, з розширеннями .c та .h включені деякі часто використовувані визначення макросів, а деякі периферійні пристрої також інкапсульовані структурними змінними, такими, наприклад, як тактова частота порту GPIO. Отже, нам треба тільки налаштувати елементи структурної змінної, щоб змінити регістри конфігурації периферійних пристроїв для вибору різних функцій. На сьогодні цей метод більше ST не підтримується.

Бібліотека HAL особливо ефективна, щоби заощадити час розробки програми. Якщо стандартна бібліотека об'єднує регістри, які необхідно налаштувати для реалізації функцій, то в бібліотеці HAL деякі функції можуть навіть інтегрувати деякі специфічні функції. Іншими словами, для тієї ж функції стандартна бібліотека може описуватися декількома виразами, а бібліотеці HAL потрібний тільки один вираз. Крім того, бібліотека HAL також розв’язує проблему міграції програм.

Стандартні бібліотеки різних моделей мікросхем stm32 розрізняються. Наприклад, програми, розроблені на STM32**F4**, не можна універсально переносити на STM32**F3**. В бібліотеці HAL програма може в основному переноситися на різні контролери. Використання програмного забезпечення STM32 CubeMX, розробленого компанією ST, яке може безпосередньо генерувати увесь файл проєкту з використанням бібліотеки HAL за допомогою функції графічної конфігурації, що дозволяє швидко і зручно стартувати проєкт, але викликає низьку ефективність реалізації.

Рекомендується завжди починати працювати, обмежуючись CMSIS, і лише розібравшись, для спрощення життя, починати використовувати SPL або HAL.

Розглянемо особливості кожної з бібліотек детально.

2.2. Бібліотека CMSIS

Стандарт програмного інтерфейсу мікроконтролера ARM® Cortex™ (Cortex Microcontroller Software Interface Standard, CMSIS) – це не залежний від виробника рівень апаратної абстракції процесорів серії Cortex-M і визначає інтерфейси налагоджувачів. ARM активно працює над тим, щоб стандартизувати інфраструктуру програмного забезпечення серед виробників MCU. CMSIS може забезпечити послідовний і простий програмний інтерфейс з процесором і периферійними пристроями, тим самим спрощуючи повторне використання програмного забезпечення, скорочуючи процес навчання для розробників-початківців мікроконтролерів і час виведення на ринок нових пристроїв.

Ядра різних виробників (FSL, ST, Energy Micro та ін.) використовують Cortex-M, але дизайн, інтерфейс і регістри периферійних пристроїв цих мікроконтролерів відрізняються. Мета CMSIS – дозволити мікроконтролерам Cortex-M різних виробників досягти певної міри узгодженості, принаймні на рівні ядра, і підвищити ефективність міграції програмного забезпечення.

CMSIS складається з таких компонентів:

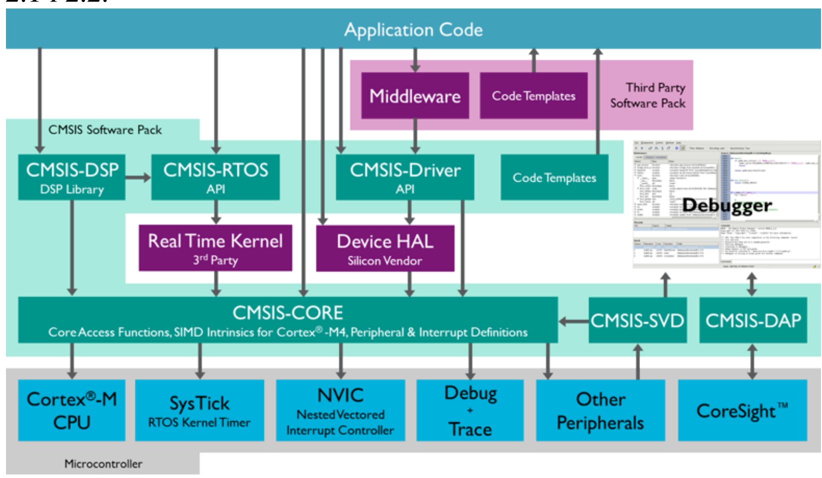
- CMSIS-CORE: API для процесора Cortex-M і периферійних пристроїв. Він забезпечує

стандартизований інтерфейс для Cortex-M0/3/4/7 і регістрами периферійних пристроїв;

- 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: із фіксованою точкою (дробові q7, q15, q31) і з плаваючою точкою одинарної точності (32-розрядні). Бібліотека доступна для Cortex-M0, Cortex-M3 і Cortex-M4;
- CMSIS-RTOS API: загальний API для операційних систем реального часу для управління потоками, ресурсами і часом. 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-RTOS безпосередньо зв'язує код застосунку користувача і стороннє ядро RTOS. Дизайн переноситься між різними RTOS або коли різні MCU Cortex переносяться безпосередньо (якщо обоє RTOS реалізують CMSIS-RTOS);
- 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 проєктів. 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 із периферійними регістрами та визначеннями переривань;
- CMSIS-DAP: порт доступу для налагодження Debug Access Port (DAP). Стандартизоване програмне забезпечення для налагоджувального блока, який підключається до порту доступу до налагодження CoreSight. CMSIS-DAP поширюється як окремий пакет і добре підходить для інтеграції на платах розробки.

Отже, CMSIS – це стандартна для всіх кортексів бібліотека, єдиний стандарт описів ресурсів. Завдяки їй легко переносити код на різні ARM Cortex мікроконтролери різних виробників (STM, NXP, TI, Cypress, Nuvoton, Fujitsu, Microchip, Toshiba, Freescale та ін.). Периферія у усіх ARM різна, навіть в межах однієї лінійки, але доступ до регістрів периферії з C стандартизований і описаний в CMSIS. Такі блоки Cortex-M, як ядро, контролер переривань і системний таймер, однакові в усіх CMSIS і при перенесенні з контролера на контролер правки не вимагає взагалі. Повний документ CMSIS можна завантажити з вебсайту ARM. Структура бібліотеки CMSIS показана на рис. 2.1 і 2.2.



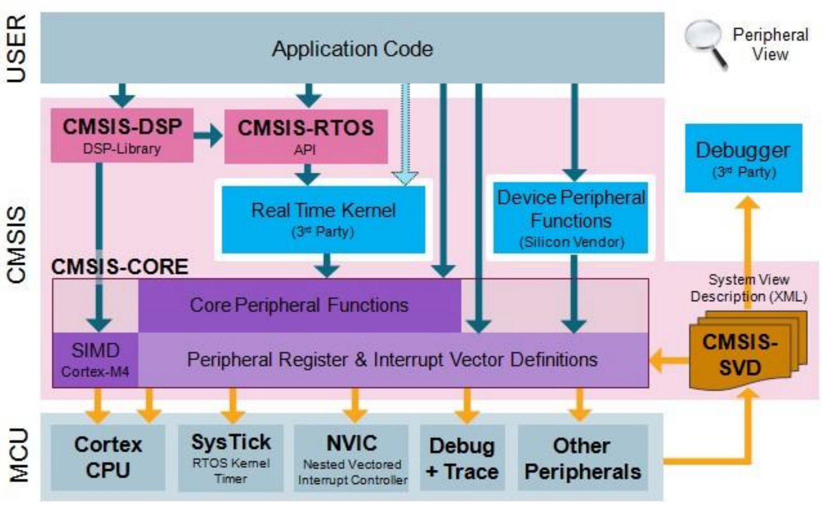


Рис. 2.1. Структура бібліотеки CMSIS

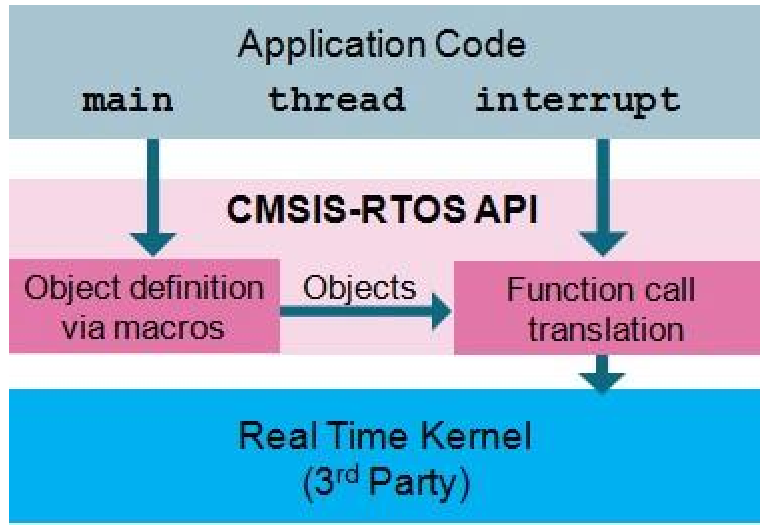


Рис. 2.2. Зв’язок коду користувача і ОС

Бібліотека складається з кількох файлів. Опис ядра стандартний для всіх Cortex-M3 і складається з двох файлів core_cm3.c та core_cm3.h, які забезпечують:

- доступ до ядра Cortex-M і його устаткування: NVIC та ін.;
- доступ до регістрів процесора Cortex-M і основних периферійних функцій.

Опис конкретного контролера (сімейства точніше), лежить в CMSIS бібліотеці конкретного сімейства і завантажується з офіційного сайту виробника контролера:

- stm32fxxx.h – заголовний файл з описом периферії, а також структури доступу до неї. Усе найпотрібніше і важливіше лежить тут;
- вказування номера переривання (відповідно до файлу запуску);
- визначення периферійного регістра (базова адреса і розташування регістра);
- функції для управління іншими унікальними функціями мікроконтролера (опціонально);
- system_stm32fxxx.h – заголовні файли для функцій CMSIS та system_stm32fxxx.c – спеціальний системний файл (функції CMSIS). Функцій небагато (SystemInit, SystemCoreClockUpdate, SystemCoreClock), стосуються вони тільки стартової ініціалізації периферії контролера і роботи з налаштуваннями тактової частоти. Вони стандартні для всіх CMSIS для M3 сімейства;
- функція SystemInit() використовується для ініціалізації мікроконтролера;
- Function void SystemCoreClockUpdate(void) використовується для отримання тактової частоти ядра;
- SystemCoreClock, це значення представляє системну тактову частоту;
- інші функції мікроконтролера (за бажанням);
- startup_stm32fxxx.s – код запуску компилятора (збірка або C);
- список обробників переривань, призначених для мікроконтролера (відповідно до файлу заголовка);

- слабка функція обробника переривань за умовчанням (може бути перевизначена призначеним для користувача кодом).

У файлі stm32fxxx.h усі регістри периферії пов' язані в таку конструкцію. Спочатку описані типи:

```
typedef struct {
__IO uint16_t CR1;
uint16_t RESERVED0;
__IO uint16_t CR2;
uint16_t RESERVED1;
__IO uint16_t OAR1;
uint16_t RESERVED2;
__IO uint16_t OAR2;
uint16_t RESERVED3;
__IO uint16_t DR;
uint16_t RESERVED4;
__IO uint16_t SR1;
uint16_t RESERVED5;
__IO uint16_t SR2;
uint16_t RESERVED6;
__IO uint16_t CCR;
uint16_t RESERVED7;
__IO uint16_t TRISE;
uint16_t RESERVED8;
} I2C_TypeDef;

typedef struct {
__IO uint32_t CRL;
__IO uint32_t CRH;
__IO uint32_t IDR;
__IO uint32_t ODR;
__IO uint32_t BSRR;
__IO uint32_t BRR;
__IO uint32_t LCKR;
} GPIO_TypeDef;
…
```

Таких і схожих записів більше половини файлу. Рекомендується відкрити цей файл яким-небудь текстовим редактором (наприклад - NotePad) і ознайомитися з його наповненням. Знадобиться для роботи з різною периферією щоб знати як звуться ті чи інші групи периферії.

Далі йде прив'язка цих типів до конкретних адрес. Спочатку задається базова адреса, звідки йде простір адрес IO:

```
#define PERIPH_BB_BASE((uint32_t)0x42000000)
    /*!< Peripheral base address in the alias region */
#define SRAM_BB_BASE((uint32_t)0x22000000)
    /*!< SRAM base address in the alias region */
#define SRAM_BASE((uint32_t)0x20000000)
    /*!< SRAM base address in the bit-band region */
#define PERIPH_BASE((uint32_t)0x40000000)
    /*!< Peripheral base address in the bit-band region */
#define FSMC_R_BASE((uint32_t)0xA0000000)
    /*!< FSMC registers base address */
```

і потім визначаються базові адреси для конкретної периферії. Записи ось такого виду:

```
/*!< Peripheral memory map */
#define APB1PERIPH_BASE        PERIPH_BASE
#define APB2PERIPH_BASE        (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE         (PERIPH_BASE + 0x20000)

#define TIM2_BASE (APB1PERIPH_BASE + 0x0000)
#define TIM3_BASE (APB1PERIPH_BASE + 0x0400)
#define TIM4_BASE (APB1PERIPH_BASE + 0x0800)
#define TIM5_BASE (APB1PERIPH_BASE + 0x0C00)
#define TIM6_BASE (APB1PERIPH_BASE + 0x1000)
#define TIM7_BASE (APB1PERIPH_BASE + 0x1400)
#define TIM12_BASE      (APB1PERIPH_BASE + 0x1800)
#define TIM13_BASE      (APB1PERIPH_BASE + 0x1C00)
#define TIM14_BASE      (APB1PERIPH_BASE + 0x2000)
...
```

Їх там дуже багато – уся периферія яка є в STM32 сімействі. Після присвоєння баз відбувається створення безпосередньо покажчиків на структури до яких ми і звертатимемося. Йдуть записи такого вигляду:

```
#define TIM2  ((TIM_TypeDef *) TIM2_BASE)
#define TIM3  ((TIM_TypeDef *) TIM3_BASE)
#define TIM4  ((TIM_TypeDef *) TIM4_BASE)
#define TIM5  ((TIM_TypeDef *) TIM5_BASE)
#define TIM6  ((TIM_TypeDef *) TIM6_BASE)
#define TIM7  ((TIM_TypeDef *) TIM7_BASE)
```

Як бачимо, тут уже фігурують імена периферійних блоків із прив'язкою покажчика до базової адреси блока (а вони всі, у свою чергу, йдуть як зміщення від базової адреси IO простору). Наприклад, загальна для усіх портів GPIO структура типу:

```
typedef struct {
    __IO uint32_t CRL;
    __IO uint32_t CRH;
```

```
    __IO uint32_t IDR;
    __IO uint32_t ODR;
    __IO uint32_t BSRR;
    __IO uint32_t BRR;
    __IO uint32_t LCKR;
} GPIO_TypeDef;
```

Наявні покажчики на неї, з прив'язкою до конкретних GPIO:

```
#define GPIOA    ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB    ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC    ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD    ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE    ((GPIO_TypeDef *) GPIOE_BASE)
#define GPIOF    ((GPIO_TypeDef *) GPIOF_BASE)
#define GPIOG    ((GPIO_TypeDef *) GPIOG_BASE)
```

Якщо тепер потрібно щось записати в регістр BSRR порту G, то звернення до нього запишеться у такому вигляді:

```
GPIOG->BSSR = 0x0001;
```

Не потрібно думати про те, де яка адреса і по якому зміщенню лежить. Усе згруповано і зібрано в зручні зв'язки. У різних процесорів усі імена звуться по-різному. У різних CMSIS від різних виробників є свої особливості, тобто у базовій частині вони збігаються один з одним, але далі можуть бути відмінності. Наприклад, у CMSIS від STM32 після обов'язкової частини йде довільна програма :

```
/*************** Bit definition for GPIO_CRL register ***************/
#define GPIO_CRL_MODE((uint32_t)0x33333333)
    /*!< Port x mode bits */
#define GPIO_CRL_MODE0((uint32_t)0x00000003)
    /*!< MODE0[1:0] bits (Port x mode bits, pin 0) */
#define GPIO_CRL_MODE0_0((uint32_t)0x00000001) /*!< Bit 0 */
#define GPIO_CRL_MODE0_1((uint32_t)0x00000002) /*!< Bit 1 */
#define GPIO_CRL_MODE1((uint32_t)0x00000030)
    /*!< MODE1[1:0] bits (Port x mode bits, pin 1) */
#define GPIO_CRL_MODE1_0((uint32_t)0x00000010) /*!< Bit 0 */
#define GPIO_CRL_MODE1_1((uint32_t)0x00000020) /*!< Bit 1 */

#define GPIO_CRL_MODE2((uint32_t)0x00000300)
    /*!< MODE2[1:0] bits (Port x mode bits, pin 2) */
#define GPIO_CRL_MODE2_0((uint32_t)0x00000100) /*!< Bit 0 */
#define GPIO_CRL_MODE2_1((uint32_t)0x00000200) /*!< Bit 1 */
```

Таких записів майже на пів мегабайта. Це розшифровування окремих бітів кожного з регістрів по функціях – готові бітмаски. Тобто виробник уже заздалегідь потурбувався про те, щоб нам

не довелося пам'ятати, де і які біти розміщені. Наприклад, щоби виставити для порту GPIOG.1 біт режиму MODE[0], навіть біти регістра CRL не доведеться рахувати. Робимо так:

```
GPIOG->CRL |= GPIO_CRL_MODE1_0;
```

Причому ім'я дефайну (визначення) вже містить повний шлях запису.

2.3. Стандартна бібліотека периферії STM32 SPL

Standard Peripherals Library (SPL) – бібліотека від STMicroelectronics для їхніх STM32 (для нових мікроконтролерів не підтримується). Базується на CMSIS, енкапсулює роботу з окремими регістрами, замість безпосереднього звертання до них, спеціальні структури заповнюються потрібними значеннями і передаються функціям, що виконують усю роботу (рис. 2.3).

Обидві бібліотеки постачаються у вигляді джерельних кодів (sources), оскільки значно простіше їх скомпілювати під конкретний контролер, ніж тримати цілий набір можливих скомпільованих варіантів. Часто, для зручності, файли бібліотек копіюються в папку проєкту (деякі IDE автоматизують цей процес, чимало IDE містять її в комплекті) (рис.2.4).

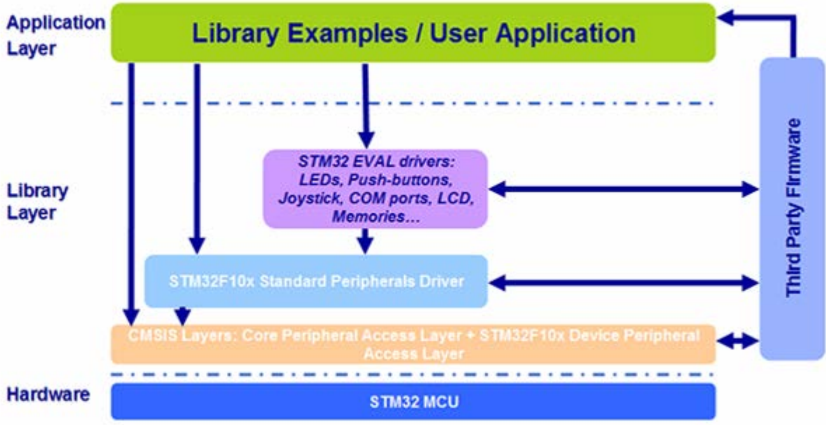


Рис. 2.3. Рівні взаємодії додатка з бібліотекою

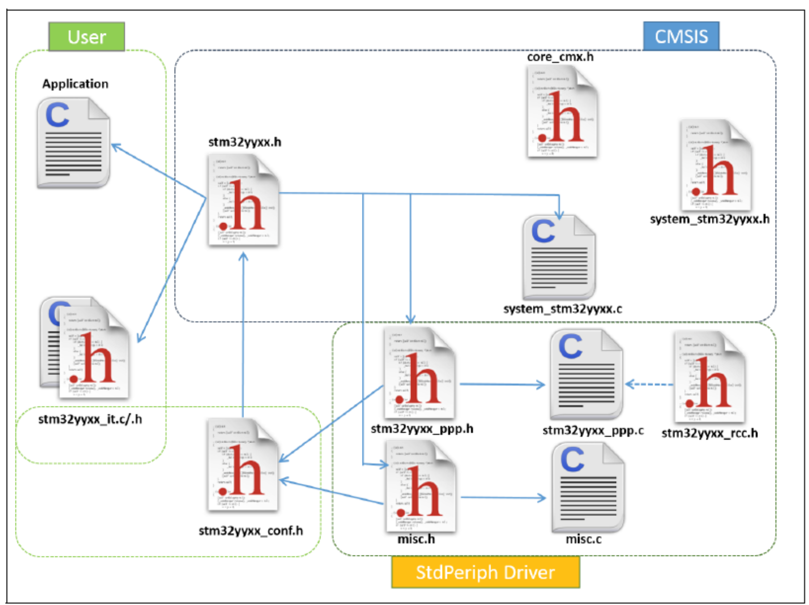


Рис. 2.4. Модель взаємодії бібліотеки із додатком

STMicroelectronics для полегшення праці розробників надає безкоштовні стандартні бібліотеки периферії для своїх мікроконтролерів і, зокрема, для сімейства STM32. Структура бібліотеки зображена на рис. 2.5.

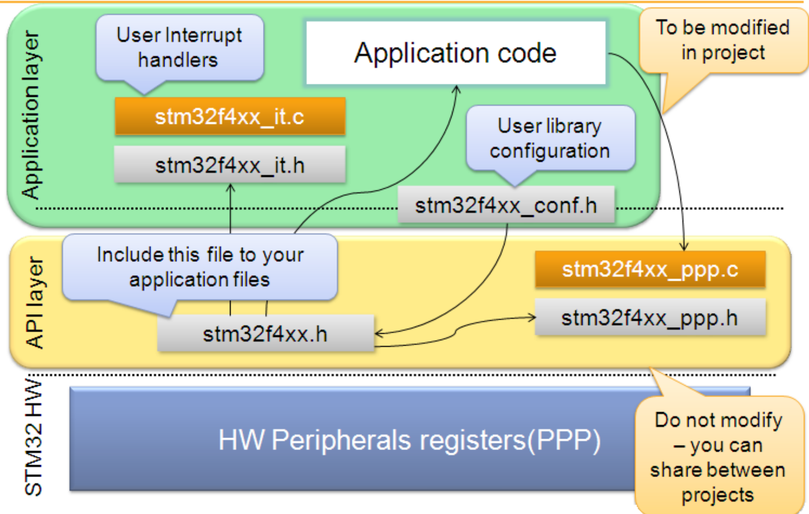


Рис. 2.5. Функціональна структура стандартної бібліотеки периферії

Стандартна бібліотека периферії написана відповідно до стандарту ANSI C і може використовуватися з будь-яким компілятором. Структура бібліотеки складається з двох взаємодоповнюючих складових.

Перша складова – заголовні файли і файли реалізації всієї периферії мікроконтролерів STM32 – STM32F10x_StdPeriph_Driver. Уся функціональність периферійних модулів описана в заголовних файлах і файлах реалізації. Наприклад, для портів введення-виведення це два файли – stm32f10x_gpio.h і stm32f10x_gpio.c.

Друга складова – заголовні файли і файли реалізації самого ядра ARM Cortex-M3 від компанії ARM – CMSIS (див. попередній підпункт). Ядро ARM Cortex-M3 виходить за рамки звичайного поняття ядра мікроконтролера і є міні-мікроконтролером з периферією – вбудовані системний таймер, контролер переривань і так далі.

Для використання стандартної бібліотеки периферії необхідно у файл основної програми (зазвичай це main.c)

включити файл #include «stm32fxxx.h» і прописати певні константи у властивостях проєкту. Бібліотекою надаються також три файли, доступні для модифікації користувачем – файл конфігурації бібліотеки stm32fxxx_conf.h і файли переривань stm32fxxx_it.h і stm32fxxx_it.c. Для використання певних модулів периферії в проєкт необхідно додати файли реалізації і конфігурувати файл stm32fxxx_conf.h. Під конфігурацією файлу stm32fxxx_conf.h мається на увазі розкоментовування рядків із назвою периферійного модуля, передбачуваного для використання. У нашому випадку це рядок #include «stm32fxxx_gpio.h». Обробка переривань відбувається в заголовному файлі і файлі реалізації stm32fxxx_it.h і stm32fxxx_it.c. Функції обробників переривання не повинні містити параметрів – void function(void). Якщо подивитися файл stm32fxxx_it.h, то можна побачити, що в ньому вже написані порожні обробники переривання, але тут є не усі функції. Додаткові імена функцій обробників переривання – це фактично адреси обробників переривання. Їх доведеться прописувати вручну, і ці імена вже оголошені у файлі початкової ініціалізації.

Уся периферія описана в структурах даних мови Сі, які використовуються для конфігурації периферійного модуля. Функції і константи для периферійних модулів розпочинаються з префіксів, що збігаються з ім'ям периферійного модуля. Наприклад, імена функцій для портів введення-виведення – GPIO_Init(), GPIO_SetBits(), GPIO_ReadInputData() і константи – GPIOA, GPIO_Speed_50MHz, GPIO_Pin_0.

Для конфігурації периферійного модуля необхідно заповнити всі поля структури і далі передати структуру функції ініціалізації периферійного модуля. Наприклад, для ініціалізації портів введення-виведення необхідно оголосити й заповнити поля структури GPIO_InitTypeDef, а відтак передати оголошене ім'я функції ініціалізації GPIO_Init(…,…). Приклад:

Спершу обмежимося засобами CMSIS:

```
/* Файл із різноманітними оголошеннями -- іменами регістрів,
бітмасками, адресами відображення в пам'яті тощо */
#include <stm32f10x.h>

int main() {
```

```
 /* Вмикаємо тактування порту С в регістрі APB2ENR підсистеми
RCC (Reset and Clock Control): встановлюємо біт
RCC_APB2ENR_IOPCEN */
RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;

 /* Піни 0-7 порту керуються регістром CRL, 8-15 -- CRH */
 /* Режим: вивід, Push-Pull, біти CNF8 – нульові, очищаємо*/
GPIOC->CRH &= ~GPIO_CRH_CNF8;
GPIOC->CRH &= ~GPIO_CRH_CNF9;
 /* Частота оновлення – 2МГц, біти MODE - 10b */
 /* Очищаємо MODE0 для PC8 */
GPIOC->CRH |= GPIO_CRH_MODE8_0;
 /* Встановлюємо MODE1 для PC8 */
GPIOC->CRH &= ~GPIO_CRH_MODE8_1;
 /* Очищаємо MODE0 для PC9 */
GPIOC->CRH |= GPIO_CRH_MODE9_0;
 /* Встановлюємо MODE1 для PC9 */
GPIOC->CRH &= ~GPIO_CRH_MODE9_1;

 /* Встановлюємо 1 на пінах 8 і 9 – засвічуємо діоди */
GPIOC->BSRR = GPIO_BSRR_BS8;
GPIOC->BSRR = GPIO_BSRR_BS9;

 /* Зациклюємо код */
while (1) {
    __NOP();
    }
}
```

Тепер повторимо те ж саме, користуючись SPL:

```
 /* Файл із різноманітними оголошеннями – іменами регістрів,
бітмасками, адресами відображення в пам'яті тощо */
#include <stm32f10x.h>
 /* Функції керування тактуванням (підсистема RCC – Reset and Clock
Control) Частина SPL */
#include <stm32f10x_rcc.h>
 /* Функції роботи з GPIO, частина SPL */
#include <stm32f10x_gpio.h>

int main() {
 /* Вмикаємо тактування порту С */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

 /* Готуємо структуру опису порту: */
GPIO_InitTypeDef LEDS_gpio_conf;
 /* Режим: вивід, Push-Pull */
LEDS_gpio_conf.GPIO_Mode = GPIO_Mode_Out_PP;
```

```
 /* Піни: 8 і 9 (світлодіоди) */
LEDS_gpio_conf.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9;
 /* Частота оновлення: 2МГц */
LEDS_gpio_conf.GPIO_Speed = GPIO_Speed_2MHz;

 /* Ініціалізуємо порт C згідно заданої конфігурації */
GPIO_Init(GPIOC, &LEDS_gpio_conf);

 /* Встановлюємо 1 на пінах 8 і 9 -- засвічуємо світлодіоди */
GPIO_SetBits(GPIOC, GPIO_Pin_8 | GPIO_Pin_9);

 /* Зависаємо */
while (1) {
    __NOP();
    }
}
```

Усі доступні функції для роботи з периферійним модулем можна подивитися в довідці на бібліотеку або в заголовному файлі. Для портів введення-виведення це файл stm32fxxx_gpio.h. Більшість периферійних модулів мають однаковий набір функцій, наприклад (PPP – ім'я периферійного модуля):

- PPP_DeInit(…) — встановлення всіх регістрів в початковий (після скидання) стан;
- PPP_Init(…) — встановлення параметрів через структури даних;
- PPP_Cmd(ENABLE/DISABLE) — дозвіл/заборона роботи (не тактування!);
- PPP_ITConfig(…) — конфігурація джерел переривань;
- PPP_GetFlagStatus(…) — читання прапорів периферійного модуля;
- PPP_ClearFlag(…) — очищення прапорів периферійного модуля;
- PPP_ClearITPendingFlag(…) — скидання прапора переривання.

Для більшої наочності коду і скорочення його візуального розміру стандартні типи даних передвизначені у файлі stm32fxxx_type.h, наприклад:

- u8 — unsigned char;
- u16 — unsigned short;
- RESET/SET;
- FALSE/TRUE;

- DISABLE/ENABLE.

Стандартну бібліотеку периферії можна завантажити з офіційного сайту фірми виробника мікроконтролерів. Наприклад, для STM32 структура пакета бібліотеки така:

- Libraries:
  - CMSIS — бібліотека ядра ARM Cortex-M3;
  - STM32F10x_StdPeriph_Lib_V3.4.0LibrariesSTM32F10x_StdPeriph_Driver — бібліотека периферії STM32.
- Project:
  - STM32F10x_StdPeriph_Lib_V3.4.0ProjectSTM32F10x_StdPeriph_Examples — початкові файли прикладів периферії STM32;
  - STM32F10x_StdPeriph_Lib_V3.4.0ProjectSTM32F10x_StdPeriph_Template — шаблон «порожнього» проєкту для оцінних плат STM3210xx — EVAL.
- Utilities — драйвера для налагоджувальних плат STMicroelectronics.
- STM32F10x_StdPeriph_Lib_V3.4.0stm32f10x_stdperiph_lib_um.chm — файл довідки.

2.4. Бібліотека HAL та LL

STM32Cube Hardware Abstraction Layer (HAL), вбудоване програмне забезпечення рівня абстракції STM32, що забезпечує максимальну переносимість коду програм мікроконтролера STM32. HAL доступний для всіх периферійних пристроїв.

HAL – набір абстрактних драйверів, що забезпечують максимальну кросплатформність серед контролерів STM32. При цьому деякі драйвери абсолютно універсальні (підходять до всіх контролерів STM32), а частина застосовна тільки до окремих лінійок з відповідними периферійними блоками (наприклад, блоками шифрування). Головні переваги HAL є такі:

- максимальна кросплатформність;
- функціональна орієнтованість. Ці драйвери спрямовані не так на роботу з окремими блоками контролера, як на виконання конкретних завдань. Це дає можливість працювати не з регістрами, а з осмисленими функціями;
- непотрібне глибоке знання архітектури мікроконтролера.

Воднораз, у HAL є й недоліки:

- значний обсяг коду;
- недостатня оптимізація виконання завдань;
- порівняно мала швидкодія.

Якщо ці недоліки критичні, слід використовувати LL-драйвери.

Low Layer APIs (LL) – апаратно залежні драйвери, що дозволяють безпосередньо працювати з периферією контролера, у тому числі – використовувати inline-функції та виконувати атомарний доступ до регістрів. Такий підхід не вимагає значних витрат пам'яті, функції виходять максимально короткими та ефективними за швидкістю. Очевидними недоліками LL-драйверів є зниження сумісності коду під час переходу від одного контролера до іншого та необхідність глибокого знання особливостей архітектури контролера.

API низького рівня (LL), що пропонують швидкий, легкий, орієнтований на експертів рівень, ближчий до апаратного забезпечення, ніж HAL. API LL доступні лише для набору периферійних пристроїв.

HAL і LL доповнюють один одного і охоплюють широкий спектр вимог до застосування:

• HAL пропонує високорівневі API, орієнтовані на функції, з високим рівнем переносимості. Вони «приховують» мікроконтролер і периферійну складність для кінцевого користувача.

• LL пропонує API низького рівня на рівні реєстрів, з кращою оптимізацією, але меншою переносимістю. Вони вимагають глибоких знань мікроконтролера та специфікацій периферійних пристроїв.

Вихідний код драйверів HAL і LL розроблено в строгому стандарті ANSI-C, що робить його незалежним від інструментів розробки. Він повністю задокументований і відповідає MISRA-C 2004.

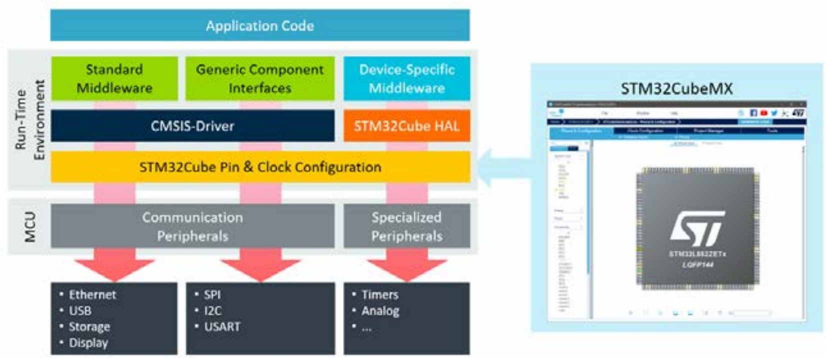


Рис. 2.6. Рівні взаємодії додатка з бібліотекою

Використання драйверів пристроїв STM32 HAL може бути корисним у багатьох ситуаціях і допомогти скоротити час розробки. Особливо для проєктів, які підпадають під категорію «підтвердження концепції».

Однак API помірного високого рівня, які надає HAL, можуть мати додаткові функції, ніж те, що вам може знадобитися в деяких ситуаціях. Отже, ви використовуєте більше пам’яті та виконуєте деякі завдання трохи повільніше через накладні витрати на функції, вбудовані в бібліотеки.

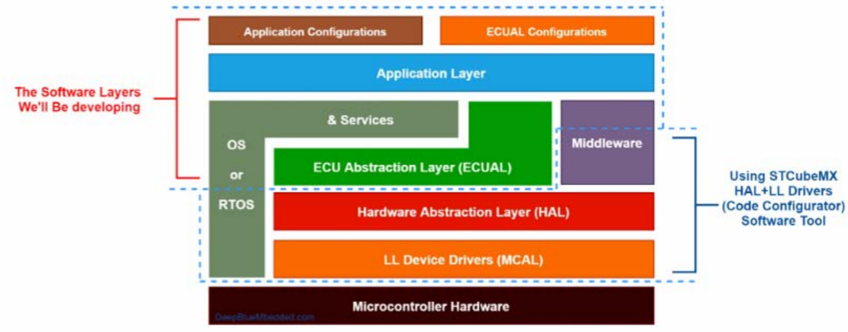


Рис. 2.7. Багаторівнева архітектура вбудованого програмного забезпечення STM32

Ви можете використовувати драйвери LL і оптимізувати більше на рівні реєстру, щоб поліпшити використання пам’яті

або швидкість виконання. Однак програму в кінці не можна буде легко перенести на кілька цілей.

Щоб досягти можливості реконфігурації, використовують те, що називається «Конфігурація зв'язування». І щоб досягти вимог щодо портативності, драйвери мають бути повністю засновані на драйверах STM HAL+LL, які мають уніфіковані API для всіх мікроконтролерів STM32.

1.2.1 Огляд

Драйвери низького рівня (LL) є частиною мікропрограми HAL STM32Cube і розроблені, щоб запропонувати швидкий легкий експертно-орієнтований рівень, який ближче до апаратного забезпечення, ніж HAL. На відміну від HAL, API LL не надаються для периферійних пристроїв, де оптимізований доступ не є ключовою функцією.

Драйвери низького рівня (LL) розроблені, щоб запропонувати:

• Набір функцій для ініціалізації периферійних основних функцій відповідно до параметрів, указаних у структурах даних;

• Набір функцій, які використовуються для заповнення структур даних ініціалізації значеннями скидання кожного поля;

• Функції для виконання периферійної деініціалізації (периферійні регістри відновлюються до значень за замовчуванням);

• Набір вбудованих функцій для прямого та атомарного доступу до регістру;

• Повна незалежність від HAL, оскільки драйвери LL можна використовувати як в автономному режимі (без драйверів HAL), так і в змішаному режимі (з драйверами HAL);

• Повне охоплення підтримуваних периферійних функцій.

Драйвери низького рівня надають апаратні послуги на основі доступних функцій у периферійних пристроях STM32. У таблиці 2 наведено вбудовані периферійні пристрої STM32, які охоплюються нижчим рівнем.

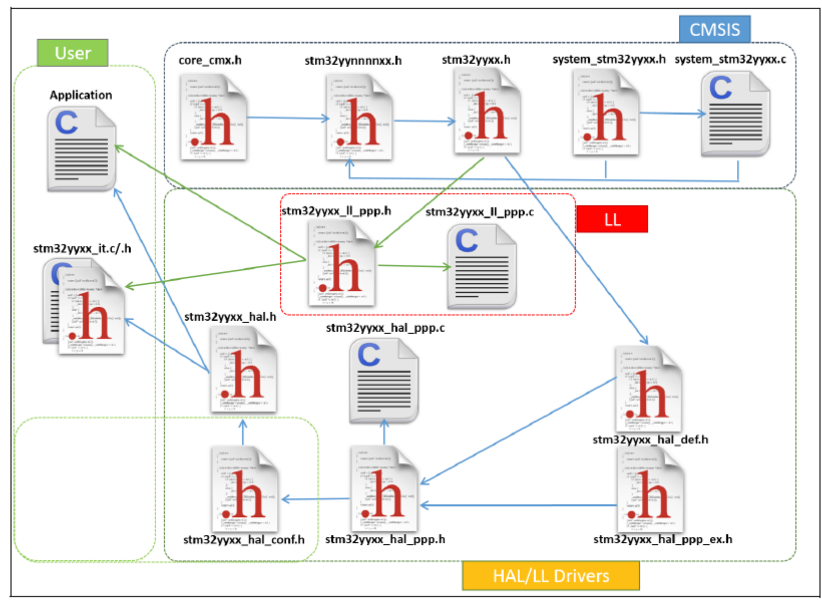


Рис. 2.8. Модель взаємодії бібліотеки з додатком

API LL точно відображають апаратні можливості та забезпечують одноразові операції. Операції мають бути викликані відповідно до моделі програмування, описаної в довідковому посібнику з лінії мікроконтролера.

Таблиця 2.1

LL підтримувана перферія

Периферія (Peripherals)		**STM32Cube підтримка**
Системна (System)	FLASH	Ні
	EXTI	Так
	GPIO	Так
	DMAXs	Так
	PWR	Так
	RCC	Так
	Cortex	Так
	SYSCFG	Так
	NVIC (покривається в	Ні (наразі

Периферія (Peripherals)		STM32Cube підтримка
	STM32 SPL драйвером misc.h/.c)	перекривається CMSIS)
Аналогова (Analog)	ADC	Так
	DAC	Так
	COMP	Так
	OPAMP	Так
	DFSDM	Ні
Таймери (Timers)	RTC	Так
	TIM	Так
	LPTIM	Так
	HRTIM	Так
	WWDG	Так
Криптографія (Cryptography)	CRC	Так
	CRYP	Ні
	HASH	Ні
	RNG	Так
Базові з'єднання (Basic connectivity)	I2C	Так
	UART/USART/LPUART	Так
	SWPMI	Так
	SPI / I2S	Так
	SDMMC (SDIO)	Ні

Як наслідок, служби LL не реалізують жодної обробки та не потребують додаткових ресурсів пам'яті для збереження своїх станів, лічильників або покажчиків даних: усі операції виконуються шляхом зміни вмісту пов'язаних периферійних регістрів.

Усі драйвери нижнього рівня викликаються через їхні фізичні екземпляри (структури периферійних регістрів, відображені в базових регістрах периферійних пристроїв) і надаються в одному модулі для кожного фізичного периферійного пристрою, розташованого в окремому файлі заголовка.

Завдяки пакету STM32Cube існують три категорії для програми STM32Cube:

• Програма HAL: програма заснована виключно на драйверах HAL.

• Програма LL: оскільки драйвери LL автономні. Користувач може розробляти свою програму, використовуючи лише драйвери LL, звертаючись до них лише зі своїх вихідних файлів. Зауважте, що драйвери LL не можуть включати один одного, вони мають включати лише файл пристрою CMSIS. Таким чином, файл конфігурації не потрібен, і користувач повинен включити використані драйвери до файлу точки входу в своїй програмі.

• Додаток MIX: додаток, у якому користувач звертається до драйверів HAL і LL і використовує обидва їхні API для розробки свого коду. Драйвери незалежні один від одного і немає ніякого втручання в підходи їх включень.

Таблиця 2.2

STM32Cube LL опис застосовуваних файлів

Ім’я файлу		**Опис**
HAL	stm32yyxx_hal.c	Загальна частина ініціалізації HAL
	stm32yyxx_hal.h	Файл заголовка загальної частини ініціалізації HAL
	stm32yyxx_hal_ppp.c	C-файл джерела драйвера PPP. Драйвер PPP є окремим модулем, для використання в проєкті користувач повинен увімкнути відповідне визначення USE_HAL_PPP_MODULE у файлі конфігурації
	stm32yyxx_hal_ppp.h	Файл заголовка драйвера PPP
	stm32yyxx_hal_ppp_ex.h/.c	Файли розширення стандартного набору API у драйвері
	stm32yyxx_def.h	Загальні ресурси HAL, такі як загальні оператори визначення, перерахування, структури та макроси
LL	stm32yyxx_II_ppp.h/.c	Вихідний h файл драйвера нижнього рівня PPP. Драйвер PPP нижнього рівня є автономним модулем. Для використання програма повинна

Ім’я файлу		Опис
		містити stm32yyxx_ll_ppp.h
CMSIS	stm32yyxx.h	Файл заголовка периферійного рівня доступу серії CMSIS Cortex-Mx STM32yyxx
	stm32yynnnnxx.h	Файл заголовка периферійного рівня доступу пристрою CMSIS Cortex-Mx STM32yyxx
	System_stm32yyxx.h/.c	Файл містить API, які викликаються під час запуску відразу після скидання та перед переходом до основної програми

На відміну від драйверів HAL, драйвери низького рівня побудовані не на моделі процесу, а на простих операціях доступу до регістрів. Таким чином, рівень низького рівня не має файлу конфігурації.

1.2.3 Визначення рівнів API і класифікація.

Метою драйверів нижнього рівня є забезпечення рівня API абстракції, який охоплює API фрагментів STM32 і функції драйверів стандартної периферії.

Драйвери нижнього рівня забезпечують додатковий набір основних API, що дозволяє налаштувати або замінити процеси високого рівня.

Кожен периферійний драйвер нижнього рівня має охоплювати такі три рівні API:

• Рівень 1: LL_PPP_WriteReg() / LL_PPP_ReadReg() (перенаправлення операцій регістрів CMSIS).

• Рівень 2: API одноразових операцій (атомні), які сортуються так:

– Керування активацією/дезактивацією периферійних пристроїв: увімкнення або вимкнення периферійного блока, підблока або пов’язану функцію. Приклад: LL_PPP_Disable(PPPx);

– Керування периферійними функціональними операціями: запустіть або запустіть периферійну операцію, або переведіть периферійний пристрій у функціональний стан. Приклад: LL_PPP_Action();

– Допоміжні операції. Приклад: IS_PPP_State(PPPx);

– Управління спеціальними прапорами переривань і стану: обробка операцій прапорів стану та реєстрації (отримання, очищення, увімкнення, вимкнення) для окремого елемента. Приклад: LL_PPP_ClearFlag_XX().

Ці API надаються в «stm32yyxx_ll_PPP.h».

• Рівень 3: Глобальна конфігурація та функції ініціалізації, які охоплюють повні автономні операції з відносними периферійними регістрами, які надаються в «stm32yyxx_ll_PPP.c».

Рівень апаратної абстракції HAL, що забезпечує переносимість між різними пристроями STM32 через стандартизовані виклики API.

Низькорівневі (LL) API, легкий, оптимізований, експертно-орієнтований набір API, призначений для продуктивності та ефективності під час роботи.

Колекція компонентів проміжного програмного забезпечення, як-от RTOS, бібліотека USB, файлова система, стек TCP/IP, бібліотека датчиків дотику або графічна бібліотека (залежно від серії MCU)

Бібліотека HAL (Hardware Acess Level) – це набір драйверів для роботи з периферійними вузлами мікроконтролера (USART, АЦП, SPI та ін.). Програміст працює з функціями цих драйверів, регістри рівня CMSIS він не використовує.

HAL є своєрідним проміжним рівнем між застосунком і рівнем CMSIS регістрів.

При такому підході не лише значно простіше розробляти програми, але і застосунки стають незалежними від типу мікроконтролера. Код легко переноситься на інші мікроконтролери STM32.

HAL драйвери надають програмістові набір API (інтерфейс прикладного програмування) для роботи з периферійними облаштуваннями STM32. Драйвери функціонально-орієнтовані, а не орієнтовані на конкретні периферійні вузли.

Наприклад, коли ми працюємо з послідовним портом UART, нам часто необхідно використовувати переривання і контролер прямого доступу до пам'яті. HAL драйвер для роботи з UART управляє всіма цими пристроями в комплексі.

Але мікроконтролер STM32 має дуже потужні ресурси. У нього 32 розрядна архітектура, великий об'єм пам'яті, висока швидкодія. Багато завдань можуть бути виконані з використанням HAL-рівня. На сучасних персональних комп'ютерах при розробці програмного забезпечення взагалі не звертають уваги на ресурсоємкість програм.

Отже потрібно вибирати, виходячи з конкретного завдання.

Є ще варіант використовувати HAL драйвери для початкової установки периферійних пристроїв. З точки зору роботи з документацією це найбільш трудомісткий процес. Велика частина регістрів встановлюється саме при ініціалізації.

До того ж процес початкового встановлення стану регістрів абсолютно не критичний до часу. Завершиться ініціалізація периферії мікроконтролера на 10 мкс раніше або пізніше після включення живлення навряд чи має значення.

Тобто можна застосовувати конфігуратор STM32CubeMX і бібліотеку HAL для ініціалізації регістрів, а при роботі програми використовувати пряме звернення до них. Або виділити окремі блоки програми, критичні до часу, і в них використовувати нижній рівень доступу до регістрів.

2.5. Приклади програм

Після конфігурування мікроконтролера за допомогою STM32CubeMX і створення проєкту режими роботи мікроконтролера та його периферійних пристроїв були задані за допомогою бібліотеки HAL.

У тексті програми зустрічаються рядки:

```
/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(LED13_GPIO_Port, LED13_Pin, GPIO_PIN_RESET);
/*Configure GPIO pin : LED13_Pin */
GPIO_InitStruct.Pin = LED13_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(LED13_GPIO_Port, &GPIO_InitStruct);
```

Неважко зрозуміти, що це встановлення початкового стану і режиму виводу LED13 за допомогою HAL бібліотеки.

Блок управління світлодіодом складається з 2 рядків виклику функцій HAL драйверів.

```
HAL_GPIO_TogglePin(LED13_GPIO_Port,LED13_Pin);
```

```
HAL_Delay(250);
```

Знову ж таки неважко зрозуміти, що це інверсія стану виводу 13 і часова затримка. Не доводиться витрачати багато часу на ознайомлення документації на STM32, розбиратися у форматах регістрів.

2.6. STM32 без HAL и SPL

Для зниження порогу входження і прискорення розробки пропонується використовувати STM32CUBE.

Для більшості проєктів не потрібні зовнішні бібліотеки і простіше використовувати звернення до регістрів мікроконтролера, використовуючи стандартну документацію.

Використання регістрів мікроконтролера – це складніший шлях, ніж використовувати обгортки із зовнішніх бібліотек, але це не завжди так.

Позначення потрібних регістрів і бітів можна узяти в 3-х документах:

1. Reference manual STM32Fxxx.pdf
2. STM32Fxxxxx.pdf
3. stm32fxxxxx.h

Цих 3-х файлів повністю вистачає для того, щоб правильно звертатися до усіх регістрів мікроконтролера.

2.6.1. Приклади ініціалізації регістрів з коментарями

Ці файли конфігурації використовуються практично без зміни тривалий час.

Звичайно ініціалізація АЦП, таймерів і ПДП дещо складніша, ніж портів, але теж – не складне завдання.

При цьому не використовуються зовнішні бібліотеки, компактніший код, більше передбачувана поведінка контролера.

2.6.2. Ініціалізація периферії

2.6.2.1. Порти

Для багатьох проєктів треба просто включати або вимикати відповідні виводи контролера і прочитувати аналогові значення.

Включення порту (1 рядок коду):

```
//ініциалізація порту A*************************
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
```

Переведення 0 піну порту А в аналоговий режим (2 рядки):

```
//PA0 - PA0/ADC1_ADC2_ADC3_IN0
// GPIO_Pin_0 порту A аналоговый вхід
```

```
GPIOA->MODER |= GPIO_MODER_MODER0_0;
GPIOA->MODER |= GPIO_MODER_MODER0_1;
```

Переведення 2 піну порту А в режим виходу (пуш-пулл) (1 рядок):

```
GPIOA->MODER |= GPIO_MODER_MODER2_0;
```

Використання альтернативних функцій теж не дуже складне. Часто мікроконтролер використовується для управління напівмостовим перетворювачем. Для цього потрібно відповідні піни порту конфігурувати як комплементарні виходи ШІМ.

Визначення виходу 8 порту А, як ШІМ вихід CH1 лічильника TIM1 (3 рядки):

```
//PA8/TIM1_CH1
// Alternate function mode
GPIOA->MODER &= ~GPIO_MODER_MODER8_0; //0
GPIOA->MODER |= GPIO_MODER_MODER8_1; //1
//GPIO alternate function high register (GPIOx_AFRL)
//AFR8[3:0] = 0001: AF1
GPIOA -> AFR[1] |= 0x00000001;
```

Визначення виходу 13 порту B, як ШІМ вихід CH1N лічильника TIM1:

```
//PB13/TIM1_CH1N
// Alternate function mode
GPIOB->MODER &= ~GPIO_MODER_MODER13_0; //0
GPIOB->MODER |= GPIO_MODER_MODER13_1; //1
//GPIO alternate function high register (GPIOx_AFRL)
//AFR13[3:0] = 0001: AF1
GPIOB -> AFR[1] |= 0x00100000;
```

2.6.2.2. АЦП.

Приклад ініціалізації АЦП для роботи в режимі DMA. У цьому режимі 4 канали АЦП2 автоматично перемикаються по колу і передають дані контролера DMA, який складає дані в масив.

```
void init_ADC1(void) {
RCC->APB2ENR |= RCC_APB2ENR_ADC1EN; //подаємо тактування
АЦП
ADC1->CR2 |= ADC_CR2_ADON; //увімкнути АЦП
ADC1->CR1 |= ADC_CR1_EOCIE;
ADC1->CR1 |= ADC_CR1_SCAN; // Bit 8 SCAN: Scan mode
ADC1->CR2 |= ADC_CR2_EOCS; //Bit 10 EOCS: End of
                                   conversion selection
ADC1->CR2 |= ADC_CR2_DMA; //Bit 8 DMA: Direct memory access
                                   mode (for single ADC mode)
ADC1->CR2 |= ADC_CR2_DDS; //Bit 9 DDS: DMA disable selection
```

(for single ADC mode

```
//Bits 23:20 L[3:0]: Regular channel sequence length (4)
//0003: 4 conversion
ADC1->SQR1 |= ADC_SQR1_L_0; //1
ADC1->SQR1 |= ADC_SQR1_L_1; //1
ADC1->SQR1 &= ~ADC_SQR1_L_2; //0
ADC1->SQR1 &= ~ADC_SQR1_L_3; //0

//Bits 4:0 SQ1[4:0]: 1st conversion in regular sequence
                                PC0/ADC1_ADC2_ADC3_IN10
ADC1->SQR3 &= ~ADC_SQR3_SQ1_0; //0
ADC1->SQR3 |= ADC_SQR3_SQ1_1; //1
ADC1->SQR3 &= ~ADC_SQR3_SQ1_2; //0
ADC1->SQR3 |= ADC_SQR3_SQ1_3; //1
ADC1->SQR3 &= ~ADC_SQR3_SQ1_4; //0

//Bits 4:0 SQ2[4:0]: 2st conversion in regular sequence
                                PC1/ADC1_ADC2_ADC3_IN11
ADC1->SQR3 |= ADC_SQR3_SQ2_0; //1
ADC1->SQR3 |= ADC_SQR3_SQ2_1; //1
ADC1->SQR3 &= ~ADC_SQR3_SQ2_2; //0
ADC1->SQR3 |= ADC_SQR3_SQ2_3; //1
ADC1->SQR3 &= ~ADC_SQR3_SQ2_4; //0

//Bits 4:0 SQ3[4:0]: 3st conversion in regular sequence
                                PC2/ADC1_ADC2_ADC3_IN12
ADC1->SQR3 &= ~ADC_SQR3_SQ3_0; //0
ADC1->SQR3 &= ~ADC_SQR3_SQ3_1; //0
ADC1->SQR3 |= ADC_SQR3_SQ3_2; //1
ADC1->SQR3 |= ADC_SQR3_SQ3_3; //1
ADC1->SQR3 &= ~ADC_SQR3_SQ3_4; //0

//Bits 4:0 SQ4[4:0]: 4st conversion in regular sequence
                                PC3/ADC1_ADC2_ADC3_IN13
ADC1->SQR3 |= ADC_SQR3_SQ4_0; //1
ADC1->SQR3 &= ~ADC_SQR3_SQ4_1; //0
ADC1->SQR3 |= ADC_SQR3_SQ4_2; //1
ADC1->SQR3 |= ADC_SQR3_SQ4_3; //1
ADC1->SQR3 &= ~ADC_SQR3_SQ4_4; //0

NVIC_EnableIRQ (ADC_IRQn);
}
```

2.6.2.3. Прямий доступ до пам'яті

Ініціалізація одного потоку ПДП для передачі даних з АЦП в масив

```
//Тактування DMA2
RCC->AHB1ENR |= RCC_AHB1ENR_DMA2EN;

//Stream3 Channel 1 DMA2 - в массив ADC2_array 2 канала

//Bits 27:25 CHSEL[2:0]: Channel selection (1)
DMA2_Stream3->CR |= DMA_SxCR_CHSEL_0; //1
DMA2_Stream3->CR &= ~DMA_SxCR_CHSEL_1; //0
DMA2_Stream3->CR &= ~DMA_SxCR_CHSEL_2; //0

//Bits 14:13 MSIZE[1:0]: Memory data size (16 bit)
DMA2_Stream3->CR |= DMA_SxCR_MSIZE_0; //1
DMA2_Stream3->CR &= ~DMA_SxCR_MSIZE_1; //0

//Bits 12:11 PSIZE[1:0]: Peripheral data size (16 bit)
DMA2_Stream3->CR |= DMA_SxCR_PSIZE_0; //1
DMA2_Stream3->CR &= ~DMA_SxCR_PSIZE_1; //0

//Bits 10 MINC: Memory increment mode
DMA2_Stream3->CR |= DMA_SxCR_MINC;

//Bits 7:6 DIR[1:0]: Data transfer direction
    (00: Peripheral-to-memory)
DMA2_Stream3->CR &= ~DMA_SxCR_DIR_0; //0
DMA2_Stream3->CR &= ~DMA_SxCR_DIR_1; //0

//Bits 4 TCIE: Transfer complete interrupt enable
DMA2_Stream3->CR |= DMA_SxCR_TCIE;

//Bits 15:0 NDT[15:0]: Number of data items to transfer
//1000 point x 4 channel
DMA2_Stream3-> NDTR = 4000;

//Bits 31:0 PAR[31:0]: Peripheral address
DMA2_Stream3->PAR = (uint32_t) &(ADC2->DR);

//Bits 31:0 M0A[31:0]: Memory 0 address
DMA2_Stream3->M0AR = (uint32_t) ADC2_array;

//Bits 0 EN: Stream enable / flag stream ready when read low
DMA2_Stream3->CR |= DMA_SxCR_EN;
```

```
NVIC_EnableIRQ (DMA2_Stream0_IRQn);
```

2.6.2.4. Таймери

Приклад конфігурації таймера з комплементарними 12-розрядними ШІМ виходами для управління 3 півмостами (3-фазний інвертор)

```
// TIM1 PWM
RCC -> APB2ENR |= RCC_APB2ENR_TIM1EN; //тактування
                                                  TIM1

TIM1->CR1 |= TIM_CR1_CMS_0; //Center-aligned mode 1
TIM1->CR1 |= TIM_CR1_ARPE;

//частота ШІМ
//прескалер 8 і період 4000 - 3000 Гц
//частота шини 108 МГц

TIM1->PSC = 8;
TIM1->ARR = 4000;

TIM1->CCR1 = 1000; //початкові значення
TIM1->CCR2 = 1000;
TIM1->CCR3 = 1000;

TIM1->CCMR1 &= ~TIM_CCMR1_OC1M_0;
TIM1->CCMR1 |= TIM_CCMR1_OC1M_1;
TIM1->CCMR1 |= TIM_CCMR1_OC1M_2; //110: PWM mode 1

TIM1->CCMR1 &= ~TIM_CCMR1_OC2M_0;
TIM1->CCMR1 |= TIM_CCMR1_OC2M_1;
TIM1->CCMR1 |= TIM_CCMR1_OC2M_2; //110: PWM mode 1

TIM1->CCMR2 &= ~TIM_CCMR2_OC3M_0;
TIM1->CCMR2 |= TIM_CCMR2_OC3M_1;
TIM1->CCMR2 |= TIM_CCMR2_OC3M_2; //110: PWM mode 1

TIM1->CCER |= TIM_CCER_CC1E; // Capture/Compare 1
                              output enable
TIM1->CCER |= TIM_CCER_CC1NE; // Capture/Compare 1
                              complementary output enable
TIM1->CCER |= TIM_CCER_CC2E; // Capture/Compare 2
                              output enable
TIM1->CCER |= TIM_CCER_CC2NE; // Capture/Compare 2
                              complementary output enable
TIM1->CCER |= TIM_CCER_CC3E; // Capture/Compare 3
                              output enable
```

```
TIM1->CCER |= TIM_CCER_CC3NE; // Capture/Compare 3
                                          complementary output enable

 //DTG[7:0]: Dead-time generator setup 1 mks
TIM1->BDTR |= TIM_BDTR_DTG_0;
TIM1->BDTR |= TIM_BDTR_DTG_1;
TIM1->BDTR |= TIM_BDTR_DTG_2;
TIM1->BDTR |= TIM_BDTR_DTG_3;
TIM1->BDTR |= TIM_BDTR_DTG_4;
TIM1->BDTR |= TIM_BDTR_DTG_5;
TIM1->BDTR |= TIM_BDTR_DTG_6;
TIM1->BDTR |= TIM_BDTR_DTG_7;

TIM1->DIER |= TIM_DIER_CC1IE; //Capture/Compare 1
                                          interrupt enable
 //TIM1->DIER |= TIM_DIER_CC2IE; //Capture/Compare 2
                                          interrupt enable
 //TIM1->DIER |= TIM_DIER_CC3IE; //Capture/Compare 3
                                          interrupt enable

TIM1->CR1 |= TIM_CR1_CEN; //Bit 0 CEN: Counter enable
TIM1->BDTR |= TIM_BDTR_MOE; //MOE: Main output enable

NVIC_EnableIRQ (TIM1_CC_IRQn); //дозволити переривання
                                          від таймера
```

2.6.2.5. Конфігурація таймера для формування переривань за часом.

Переривання від цього таймера, як правило, використовуються для оновлення даних в інтерфейсах. У цьому режимі зовнішні виходи таймера не використовуються.

```
 // TIM3 100 мс
RCC -> APB1ENR |= RCC_APB1ENR_TIM3EN; //TIM3 Timer clock
                                                      enable

TIM3->CR1 |= TIM_CR1_CEN; //Bit 0 CEN; // Counter enable
TIM3->CR1 |= TIM_CR1_ARPE; //Bit 7 ARPE; // Auto-reload preload
                                                      enable
TIM3->DIER |= TIM_DIER_UIE; //Bit 0 UIE: Update interrupt enable

TIM3->PSC = 2000;
TIM3->ARR = 5400;
NVIC_EnableIRQ (TIM3_IRQn); // дозволити переривання
                                          від таймера
```

2.7. Література до розділу

1. http://indrekis2.blogspot.com/2012/10/stm32-arm-cortex-m-stmicroelectronics.html
2. http://mypractic.ru/urok-4-rabota-s-registrami-mikrokontrollera-stm32-biblioteki-cmsis-i-hal.html
3. https://www.drive2.ru/b/2253235/
4. https://russianblogs.com/article/389975940/
5. Основи проєктування систем Інтернету речей. Периферія мікроконтролерів STM32 Конспект лекцій: Навчальний посібник.– Укладачі: Ю.О. Оникієнко, А.Р. Рижова, Електронне мережне навчальне видання, Київ, КПІ ім. Ігоря Сікорського, 2022.– Ресурс: https://ela.kpi.ua/bitstream/123456789/48924/1/Osnovy_proektuvannia_system_Internetu_rechei.pdf
6. https://siliconlabs.github.io/Gecko_SDK_Doc/CMSIS/General/html/index.html
7. https://developer.arm.com/tools-and-software/embedded/cmsis
8. https://www.digikey.com/en/maker/projects/getting-started-with-stm32-introduction-to-freertos/ad275395687e4d85935351e16ec575b1
9. https://russianblogs.com/article/21871603717/
10. http://easyelectronics.ru/arm-uchebnyj-kurs-keil-cmsis-sozdanie-proekta.html
11. http://indrekis2.blogspot.com/2012/10/stm32-arm-cortex-m-stmicroelectronics.html
12. Ді Халт: "ARM. Учебный Курс. Keil + CMSIS. Создание проєкта".– http://easyelectronics.ru/arm-uchebnyj-kurs-keil-cmsis-sozdanie-proekta.html
13. Ді Халт: "Работа с STM32F10x Standard Peripherals Library".– http://easyelectronics.ru/rabota-s-stm32f10x-standard-peripherals-library.html
14. https://www.compel.ru/lib/53954
15. https://www.st.com/resource/en/application_note/an5044-stm32-standard-peripheral-library-to-stm32cube-lowlayer-migration-stmicroelectronics.pdf
16. https://deepbluembedded.com/stm32-hal-library-tutorial-examples/
17. http://mypractic.ru/urok-4-rabota-s-registrami-mikrokontrollera-stm32-biblioteki-cmsis-i-hal.html
18. http://indrekis2.blogspot.com/2012/10/stm32-arm-cortex-m-stmicroelectronics.html
19. https://habr.com/ru/post/337622/

РОЗДІЛ 3. Основи програмування мовою високого рівня Сі

Мова програмування Сі – високорівнева абстрактна мова програмування. Перша версія мови Сі створена в середині 60-х років для розроблення операційної системи Unix в лабораторії Bell. Один із найперших розробників цієї мови, Кен Томсон, вирішив, що потрібна мова для створення більш складних мов програмування. Він назвав її "B". В процесі розвитку свого творіння Томсон постійно боровся з обмеженням ресурсів пам'яті, що зараз дуже схоже на вбудовані системи. Денніс Річі вирішив розширити мову "B" властивістю генерувати малий за об'ємом код, який зможе змагатися з кодом, написаним на асемблері. У 1973 році найважливіші властивості цієї нової мови "Сі" були отримані.

Зростаюча популярність Сі зумовлена її переносимістю. Компілятори Сі були створені для багатьох платформ, завдяки чому її популярність ще більше виросла. Найбурхливіше Сі стала використовуватися у 80-х роках, коли стала основною мовою для створення програм персональних комп'ютерів. Основною ідеєю мови Сі є робота з функціями, аргументи яких передаються як параметри.

3.1. Ввідні поняття

Передусім, розглянемо такі ввідні поняття, як коментарі, ключові слова, ідентифікатори, літерали, оператори і знаки пунктуації.

Коментар – пояснюючий текст, який при компіляції не враховується. Коментарі бувають багаторядковими (розпочинаються з символів /* і закінчуються символами */) і однорядковими (розпочинаються з символів //). Наприклад:

```
/*Багаторядковий коментар часто розміщують на початку файлу, де
він містить ім'я автора і опис программи*/
#include "MDR 32Fx.h" //Підключаємо заголовний файл MDR32Fx.h
```

Ідентифікатор – це послідовність букв, цифр і символів підкреслення «_», яка не повинна розпочинатися з цифри, використовувана для іменування різних програмних елементів,

на зразок змінних, констант, функцій, типів і так далі. Регістр букв має значення.

Ключове слово – це зарезервоване слово чіткого певного призначення. Ключові слова не можуть використовуватися як ідентифікаторів:

asm, auto; bit, bool, break; case, char, const, continue default, defined, do, double; else, enum, explicit, extern; false, float, for; goto; if, inline, int; long; register, return; short, signed, sizeof, static, struct, switch; true, typedef; union, unsigned; void; while.

Літерал – постійне значення деякого типу, використовуване у виразах. Приклади числових літералів:

10 — число 10 в десятковій формі;
0xA — число 10 в шістнадцятковій формі (префікс 0x);
0b1010 — число 10 в двійковій формі (префікс 0b);
012 — число 10 у вісімковій формі (префікс 0);
10.5 — число з плаваючою точкою;
105e-1 — число 10,5 в експоненціальній формі;
10U — беззнакова константа (суфікс U);
10L — знакова константа (суфікс L).

Символьні літерали беруться в одинарні лапки, наприклад, 'B', '*'. Для позначення недрукованих і спеціальних символів у літералах використовуються так звані escape-послідовності:

'\a' — звуковий сигнал;
'\b' — клавіша <Backspace>;
'\f' — прогін листа;
'\n '— символ переведення рядка
'\r '— повернення каретки;
'\t '— горизонтальна табуляція;
'\v' — вертикальна табуляція;
'\0'— нульовий символ;
'\\'— зворотна коса;
'\''— апостроф.

Крім того, будь-який символ можна представити за допомогою літерала по його ASCII-коду, наприклад, літерал '\t' рівнозначний '\x09' (в шістнадцядковому форматі).

Рядкові літерали обмежуються подвійними лапками, а в пам'яті зберігаються як послідовності символів, що закінчуються нульовим символом '\0'. Спеціальні символи усередині рядка повинні упереджатися зворотною косою ("\"). Приклади рядкових літералів:

"" — порожній рядок: один символ '\0';
"B" — два символи: 'B' і '\0';

"A\tB\n" — п'ять символів: 'A', табуляція, 'B', переведення рядка, '\0'.

Оператор – це символ, що вказує компілятору, які дії виконати над операндами. Деякі символи можуть трактуватися по-різному залежно від контексту. Наприклад, знак "-" може використовуватися для зміни знака числа або як оператор віднімання. Оператори, що сполучають операнди, є виразами. Вирази можуть бути поміщені в круглі дужки і відділяються один від одного символом крапки з комою (";").

Пріоритет (за убуванням) виконання операторів у виразах мови С вказаний в таблиці 1.1.

Таблиця 3.1

Пріоритет виконання операторів у мові С

<table>
<tr><th>Рівень</th><th>Оператори</th><th>Категорія</th><th>Опис</th></tr>
<tr><td rowspan="5">1</td><td>()</td><td></td><td>Круглі дужки</td></tr>
<tr><td>[]</td><td></td><td>Елемент масиву</td></tr>
<tr><td>.</td><td rowspan="2">Доступ до даних</td><td>Звернення до елемента структури, наприклад: PortInit.PORT_OE – звернення до регістру напрямку передавання даних</td></tr>
<tr><td>-></td><td>Звернення до елемента структури, визначеної вказівником, наприклад: pStruct ->x – елемент x структури на яку вказує pStruct</td></tr>
<tr><td>++, --
(постфікси)</td><td rowspan="2">Арифметичні</td><td>Оператори автоінкременту і автодекременту після того, як вираз, в якому задіяні відповідні операнди, буде обчислений. Приклади:
a = b++; рівнозначно a = b; b = b+1;
a = b--; рівнозначно a = b; b = b-1;</td></tr>
<tr><td rowspan="3">2</td><td>++, --
(префікси)</td><td>Оператори автоінкременту і автодекременту перед тим, як вираз, в якому задіяні відповідні операнди, буде обчислений. Приклади:
a = ++b; рівнозначно b = b+1; a = b;
a = --b; рівнозначно b = b-1; a = b;</td></tr>
<tr><td>!</td><td>Логічні</td><td>Логічне (унарне) заперечення</td></tr>
<tr><td>~</td><td>Порозрядні</td><td>Порозрядне заперечення</td></tr>
</table>

<table>
<tr><th>Рівень</th><th>Оператори</th><th>Категорія</th><th>Опис</th></tr>
<tr><td></td><td>&</td><td>Доступ до даних</td><td>Адреса</td></tr>
<tr><td rowspan="2">3</td><td>+, - (унарні)</td><td>Арифметичні</td><td>Зміна знака операнду</td></tr>
<tr><td>* (унарний)</td><td>Доступ до даних</td><td>Розіменування вказівника</td></tr>
<tr><td rowspan="3">4</td><td>* (бінарні)</td><td rowspan="4">Арифметичні</td><td>Множення</td></tr>
<tr><td>/</td><td>Ділення</td></tr>
<tr><td>%</td><td>Залишок від ділення</td></tr>
<tr><td>5</td><td>+,- (бінарні)</td><td>Додавання і віднімання</td></tr>
<tr><td rowspan="2">6</td><td><<</td><td rowspan="2">Порозрядні</td><td>Порозрядний зсув вліво</td></tr>
<tr><td>>></td><td>Порозрядний зсув вправо</td></tr>
<tr><td>7</td><td><, >, <=, >=</td><td rowspan="2">Порівняння</td><td>Менше, більше, менше або дорівнює, більше або дорівнює</td></tr>
<tr><td>8</td><td>= =, !=</td><td>Дорівнює, не дорівнює</td></tr>
<tr><td>9</td><td>&</td><td rowspan="3">Порозрядні</td><td>Порозрядне "І"</td></tr>
<tr><td>10</td><td>^</td><td>Порозрядне "Виключне АБО"</td></tr>
<tr><td>11</td><td>|</td><td>Порозрядне "АБО"</td></tr>
<tr><td>12</td><td>&&</td><td rowspan="2">Логічні</td><td>Логічне "І"</td></tr>
<tr><td>13</td><td>||</td><td>Логічне "АБО"</td></tr>
<tr><td>14</td><td>=</td><td>Присвоювання</td><td>Можливі також поєднання оператора присвоювання з арифметичними і порозрядними операторами, наприклад:
a+ = b; рівнозначно a = a + b;
a* = b + c; рівнозначно
a = a * (b + c); □</td></tr>
</table>

Об’єднання за деякою ознакою послідовності виразів беруться у фігурні дужки { }. Так, наприклад, позначаються границі функцій, а також блоки виразів у циклічних і умовних конструкціях.

Коли у виразі використовується оператор інкремента або декремента, то важливо звертати увагу на те, де стоять два знаки "+" або "-": перед змінною або після змінної:

a=4, b=7;

a = b++; / Цей рядок на С означає: узяти значення змінної b присвоїти його змінній, а потім додати 1 до змінної b і зберегти результат в b. Тепер a = 7; b = 8 */*

a=4, b=7;
a = ++b; / Цей рядок на С означає: узяти значення змінної b потім додати до нього 1 і зберегти результат в b і цей же результат присвоїти змінній a. Тепер a = 8, b = 8.*/*

3.2. Структура програми мовою Сі

Структура програми написаної мовою Сі має такий вигляд:

```
//Директиви препроцесора
#include "STM32Fx.h"
#define BLINK_DELAY 20000
…
//оголошення глобальних типів, змінних, констант
uint32_t cnt;
float x, y;
…
//Призначені для користувача функції
Function1
{
 //Оголошення локальних типів, змінних і констант
…
 //Оператори
 …
}
…
Function N
{
 //Оголошення локальних типів, змінних і констант
…
 //Оператори
 …
}

//Головна функція програми
int main (void)
{
 //Оголошення локальних типів, змінних і констант
…
 //Оператори
…
}
```

Програми зазвичай розпочинаються з директив процесора (які розпочинаються з символу «#»), які не є конструкціями мови Сі і обробляються до фактичної компіляції програми. Їх сенс – підстановка деякого коду в програму. Так, приміром, дуже часто використовується директива #include, яка включає у файл з початковим кодом програми текст зовнішнього заголовного файлу (з розширенням .h). Заголовні файли містять визначення глобальних типів, змінних констант і функцій.

Далі в програмі оголошуються глобальні змінні, типи, константи. Звертатися до таких змінних можна з будь-якого місця програми.

Потім описуються функції, що входять в програму. Функція є "контейнером", в якому виконується деякий фрагмент програмного коду. Використання функцій спрощує написання і відлагодження програм, оскільки в них зручно розміщувати групи операторів, що повторюються. Будь-яка програма мовою Сі містить головну функцію під назвою main( ). Ця функція виконується першою при запуску програми.

Функції користувача визначаються після директив препроцесора і глобальних оголошень типів, змінних і констант у файлі з початковим кодом або в заголовному файлі. При цьому використовується такий синтаксис:

```
Тип_значення_що_повертається Ім'я_функції (Список_параметрів)
{
//Тіло функції
}
```

Попереднього оголошення функції може не бути, але в цьому випадку вона доступна тільки всередині того файлу, в якому визначена.

Як ім’я функції можна використовувати будь-яке слово, що відображає виконувану функцією дію, при цьому ім'я не повинне розпочинатися з цифри.

Тіло функції є набором команд і операторів, розташованих між фігурними дужками *{ }*.

Функцію викликають за її ім’ям із вказанням в круглих дужках переліку передаваних параметрів (якщо їх немає, то в дужках нічого не вказується), наприклад:

```
void Function1 (int n, char c)
{
```

```
…
}

int Function2 ()
{
…
}

int main ()
{
int x;
char y;

Function1(x, y);
x=Function2();
}
```

Параметри (список параметрів), один або декілька, які можуть передаватися у функцію – це ідентифікатори, які можуть використовуватися усередині функції. Замість них підставляються відповідні значення, вказані при виклику функції (якщо у функцію передається більше за один параметр, то вони відділяються один від одного комами, як при оголошенні, так і при виклику).

Значення, передані у функцію, фактично не змінюються, а просто копіюються в параметри, які в цьому сенсі виконують роль локальних змінних. При цьому слід стежити за тим, щоб тип передаваних значень відповідав типу параметрів, оголошених в заголовку функції.

Повертати функція може тільки один параметр. Як значення, що повертається, може використовуватися ключове слово *void*. Це означає, що функція не повертає ніякого значення. Якщо функція призначена для повернення значення деякого типу, то для цього в її тілі використовують ключове слово *return*, після якого (через пропуск) вказують значення, яке повертається. При цьому всі оператори після слова *return* ігноруются, і відбувається повернення до викликаючої функції. Наприклад:

```
int power3(int n)
{
 return n*n*n;
}
void main()
{
```

```
 int x;
 x = power3(2); //x = 8
}
```

Слово *return* може також використовуватися без вказівки виразу що перевіряється. В цьому випадку воно просто означає вихід з функції.

Прототипи функцій – це попереднє оголошення тільки заголовку функції до того, як вона зустрінеться в тексті програми. Тобто в звичному варіанті функції використовуються тільки після їх визначення. Однак бувають випадки, коли функції викликають одна одну, і організувати їх "правильне" визначення неможливо. Обійти подібну проблему дозволяють прототипи функцій, які яавляють собою оголошення до визначення. Таке оголошення є тільки заголовком функції, причому в переліку параметрів вказують лише типи, без ідентифікаторів, наприклад:

```
…
int f1 (int);
void f2 (int, int);

int f1 (int x)
{
…
}

void f2 (int a, int b)
{
…
}
```

Прототипи функцій часто застосовуються в заголовних файлах, що включаються в текст програми за допомогою директиви препроцесора *#include*.

3.2.1. Класи пам'яті при оголошенні локальних змінних

Локальні змінні можуть бути оголошені всередині функцій як ті, що належать до одного з трьох класів пам'яті:

auto (значення за замовчуванням, явно можна не вказувати) – при оголошенні змінна не ініціалізується жодним значенням (значення – поточний вміст області пам'яті, яка відведена під змінну); при виході з функції змінна видаляється з пам'яті;

static – статична змінна доступна тільки в межах функції, хоча пам’ять для неї виділяється в просторі глобальних змінних; при першому звертанні до функції ініціалізується нульовим значенням, а після виходу з функції із пам’яті не видаляється (отже, при наступних звертанням до функції в ній міститься "старе" значення);

register – аналог автоматичної локальної змінної за тим вийнятком, що компілятор буде намагатися виділити для неї не область пам’яті даних, а робочий регістр мікроконтролера, що значно прискорює звернення до значення змінної.

Приклад застосування статичної змінної:

```
…
int plus5 ()
{
static int x;
return x+5;
}

void main ()
{
int y;
y = plus5(); //y=5
y = plus5(); //y=10
y = plus5(); //y=15
}
```

3.2.2. Рекурсія

Рекурсія – це виклик функцією самої себе. Приклад рекурсивного виклику:

```
void f1 (int n)
{
int x;
f1(x);
}
```

Незважаючи на складність сприйняття, рекурсія досить часто використовується в стандартних бібліотечних функціях, а також в багатьох алгоритмах сортування. Тим не менш, при програмуванні мікроконтролерів застосування рекурсії, як правило, може викликати проблеми через обмежений об’єм оперативної пам’яті (пам’яті даних). При кожному виклику рекурсивної функції частина пам’яті йде на збереження даних, які розміщуються в стек. Ці дані зберігаються там доти, поки не

буде виконано повернення з функції. Таким чином, коли рекурсивна функція знову і знову викликає саму себе, в стекові залишається дедалі менше вільної пам’яті.

Можна сказати, що використання рекурсії при програмуванні мікроконтролерів – це надійний спосіб швидкого і непередбачуваного заповнення стеку. Це часто призводить до виникнення помилкового стану, яке називається переповненням стеку.

Стек буває апаратний або програмний. Апаратний стек являє собою окрему область пам’яті на кристалі і має визначену кількість комірок. При переповненні починає записувати дані по колу, таким чином затираючи перші записані (це зазвичай адреси повернення з підпрограм). Програмний стек розміщений в ОЗП або, по іншому, в пам’яті даних. Зазвичай ця область розміщується у верхній частині пам’яті даних і росте "вниз", тоді як область змінних розміщується в нижній частині і росте "вверх". Тому, якщо об’єм даних, розміщуваних у стеку, перевищить розмір області стеку, ці дані можуть досягнути області змінних і затерти собою її значення. При цьому помилку переповнення стеку не завжди легко виявити, оскільки вона може проявлятися лише періодично.

При багатократному вкладенні функцій може виникнути така ж проблема, тобто коли, наприклад, функція f1 викликає функцію f2, яка викликає функцію f3, що викликає функцію f4 і т.д.

3.3. Типи даних, змінні, константи, структури

Тип даних визначає діапазон допустимих значень і відповідно – простір, що відводиться в пам'яті даних, для змінних, констант і результатів, повертаних функціями. До стандартних типів даних мови С відносять: *signed char; unsigned char; int; unsigned int; long; unsigned long; float; long long; unsigned long long*.

Таблиця 3.2

Стандартні типи даних мови С

Тип	Назва	Довжина, біт	Діапазон значень
bit	Біт	1	0, 1
char	Знаковий символ	8	-128 ... 127
signed char			
unsigned char	Беззнаковий символ або логічний символ		0 … 255, TRUE, FALSE
int	Ціле число	16	-32768 ... 32767
signed int	Знакове ціле число		
short int			
signed short int			
unsigned int	Беззнакове ціле число		0 … 65535
unsigned short int			
long int	Довге знакове ціле число	32	-2 147 483 648 ... 2 147 483 647
signed long int			
unsigned long int	Довге беззнакове ціле число		0...4 294 967 295
float	Дійсне число	32	-3.4E38 ... 3.4E38
double	Подвійне дійсне число (подвійної точності)	64	1.7E-308 ... 1.7E+308
long double	Довге дійсне число (підвищеної точності)	80	3.4E-4932 ... 3.4E+4932

3.3.1. Типи користувача

Мова Сі дозволяє, крім стандартних, оголошувати власні типи. Для цього використовується ключове слово *typedef*, наприлад:

typedef unsigned char byte; // оголошуємо тип byte
typedef unsigned int word; // оголошуємо тип word

Іншими словами, для оголошення користувацького типу використовується конструкція вигляду

typedef стандартний_тип ідентифікатор_користувацького_типу;

Наприклад:

*typedef signed char **int8_t**;*
*typedef signed unsigned char **unt8_t**;*
*typedef int **int16_t**;*
*typedef unsigned int **uint16_t**;*
*typedef unsigned long **uint32_t**;*
*typedef long long **int64_t**;*
*typedef unsigned long long **uint64_t**;*
*typedef struct {int quot; int rem;} **div_t**;*
*typedef struct {long quot; long rem;} **ldiv_t**;*

3.3.2. Змінні

Змінна – це іменована величина певного типу, яка може змінюватися в ході виконання програми. Для оголошення змінних (тобто, виділення для них пам'яті) в програмі на Сі використовується така конструкція:

тип_змінної идентификатор1, идентификатор2, . . .;

наприклад:

int i;//Оголошення цілочисельної змінної i
char c1, c2; //Оголошення символьних змінних c1 і c2

Для доступу до змінної в програмі використовується відповідний ідентифікатор (обов'язково після оголошення змінної). Значення, що присвоюються змінній, повинні відповідати їй за типом (чи правилам приведення типів, що розглядаються нижче).

Приклади:

i = 2;//Помилка! Змінна i ще не оголошена
int i;//Оголошення цілочислової змінної i
float f; //Оголошення дійсної змінної f
i = 2; //Змінній i присвоєно значення 2
f= 3.3 //Змінній f присвоєно значення 3.3
f= i; //Змінній f присвоєно значення змінної i
//(в даному випадку буде виконано автоматичне приведення типів, тобто f = 2.0)

За областю видимості змінні можуть бути глобальними і локальними. До глобальних змінних мають доступ усі функції програми. Такі змінні оголошуються в програмі перед

оголошенням усіх функцій. До локальних змінних має доступ тільки та функція, в якій вони оголошені.

Імена змінних мають певну область видимості, яка має на увазі, що компілятор використовує змінні відповідно до того, де вони знаходяться. Імена змінних, оголошених усередині функції, мають зону видимості, обмежену конкретною функцією. Наприклад, у кількох функціях можна оголосити змінну *int i*, яка в кожній функції не матиме ніякого зв'язку з аналогічними змінними в інших функціях. Так само і змінна, оголошена усередині блока (обмеженого фігурними дужками *{…}*), залишається локальною по відношенню до цього блока.

Глобальні змінні мають область видимості, яка починається від місця їх оголошення і триває до кінця програмного файлу. Для того щоби глобальну змінну можна було використовувати в інших файлах, її потрібно об’явити за допомогою ключового слова *extern*:

```
extern int n;
```

Об’явлену в такий спосіб змінну, перш ніж її використовувати, слід обов’язково ініціалізувати в зовнішньому файлі деяким значенням.

Використаннях змінних, об’явлених за допомогою ключового слова *extern*, часто призводить до помилок в програмах. Тому рекомендується їх застосовувати тільки тоді, коли немає іншої альтернативи і потрібно бути уважним.

3.3.3. Константи

Константа – це іменована величина певного типу, яка, на відміну від змінної, не може змінюватися в ході виконання програми, а має конкретне значення, визначене у момент оголошення. Для оголошення констант в програмі на Сі використовується наступна конструкція:

```
const тип_константи ідентифікатор = значення;
```

Наприклад:

```
Const int i=10;//Оголошення цілочислової константи i.
```

Величина, оголошена як константа, буде розміщена компілятором в пам'яті RAM, тобто в тій самій області пам'яті, де зберігаються змінні. Для доступу до константи використовується її ідентифікатор.

Приклади:

```
c ='A'; //Помилка! Константа c ще не оголошена
int i; //Оголошення цілочислової змінної i
const c ='A';//Оголошення константи c
i = 2; //Змінній i присвоєно значення 2
c = i; //Помилка! Спроба присвоїти значення константі
i = c; //Змінній i присвоєно значення константи c
/* В даному випадку буде виконано автоматичне приведення типів,
тобто i= 65 (ASCII -код символу 'A') */
```

3.3.4. Перечислювані типи

Перечислюваний тип – це оголошення списку цілочислових констант, які можна явно не ініціалізувати (в цьому випадку вважається, що перша константа в списку набуває значення 0, друга – 1 і т.д.). Для подібного оголошення використовується ключове слово enum:

```
int n;
enum {zero, one, two}; // zero=0; one=1; two=2
n=one;// n=1
```

Якщо необхідно змінити початкове значення для списку констант, то можна вказати його явно при оголошенні, наприклад:

```
enum {three=3, four, five}; // three=3; four=4; five=5
```

3.3.5. Приведення типів

Приведення типів – це примусове перетворення значення одного типу до іншого, сумісного з початковим. Це важливо при виконанні арифметичних операцій, коли отримані значення можуть виходити за допустимі межі.

Приведення типів буває явним і неявним. Неявне приведення типів використовується в операторах присвоєння, коли компілятор сам виконує необхідні перетворення без участі програміста.

Для явного приведення типу деякої змінної перед нею слід вказати в круглих дужках ім'я нового типу, наприклад:

```
int X;
int Y = 200;
char C = 30;
X = (int) C * 10 + Y; //Змінна C приведена до типу int
```

Якби в даному прикладі не було виконано явне приведення типів, то компілятор припустив би, що вираз C*10 – це восьмирозрядне множення (розрядності типу *char*) і замість

коректного значення 300 (0x12C) в стек було б поміщено урізане значення 44 (0x2C). Отже, в результаті обчислення виразу C*10+Y змінній X було б присвоєне значення 244, а не коректне 500. В результаті приведення типу змінна C розпізнається компілятором як 16-ти розрядна й описаної вище помилки не виникає.

3.3.6. Оператор sizeof

Оператор *sizeof* застосовується для обчислення розміру області пам'яті (у байтах), що відводиться під деяку змінну, результат виразу або тип. Наприклад:

```
int a;
float f;
a = sizeof (int); //a = 2
a = sizeof (f); //a = 4
f = 3.3;
a = sizeof(f + a); //a = 4, оскільки тип результату - float
```

3.3.7. Структури

Структура – це особливий тип даних, що складається з кількох різнотипних змінних (полів). Стуктури зручно використовувати, коли потрібно об'єднати кілька змінних або типів даних (однакових чи різних) у певну групу. У загальному випадку оголошення структури має такий вигляд:

```
struct ім'я_структури
    {
    тип поле_1;
    …
    тип поле_N;
    };
```

Як і будь-який інший тип, структуру можна надалі використовувати для оголошення змінних, наприклад:

```
struct MyStructure //Оголошення структури MyStructure
    {
    int Fieldl;
    char Field2;
    float Field3;
    };
struct MyStructure YourStruct, OurStruct; //Оголошення змінних YourStruct
і OurStruct типу MyStructure
```

або

```
struct { int a; long b; char c; } st;
```

Можна оголосити масив структур:

```
struct { int a; long b; char c; } st[5];
```

Допускається ініціалізація полів безпосередньо при оголошенні змінних-структур за допомогою переліку значень у фігурних дужках, наприклад:

```
struct DATE
    {
    unsigned char Day;
    unsigned char Month;
    int Year;
    }
struct DATE MyBirthday = {7, 8, 1980};
```

Для доступу до полів структури в програмі використовують запис виду *ім'я_структури.поле*. Отже, в наведеному вище прикладі структури *DATE* для ініціалізації полів можна було скористатися такими операторами:

```
MyBirthday.Day = 7;
MyBirthday.Month = 8;
MyBirthday.Year = 1980;
```

Структури, у свою чергу, можуть бути полями інших структур, наприклад:

```
struct ME
    {
    char MyName[30]; //Рядок завдовжки 30 символів - ім'я
    struct DATE MyBirthday; //Структура, що зберігає день народження
    }
struct ME MyData;
MyData.MyName = "John Smith";
MyData.MyBirthday.Day = 7;
MyData.MyBirthday.Month = 8;
MyData.MyBirthday.Year = 1980;
```

Структури можуть виступати параметрами функцій, а також поверненим результатом. Нижче наведений приклад функції, що повертає структуру типу *DATE*:

```
struct DATEJanuaryl (int CurYear)
    {
    struct DATE Jan0l;
    Jan0l. Day = 1;
    Jan0l.Month = 1;
    Jan0l.Year = CurYear;
    return JanOl;
    }
…
struct DATE BeginOfTheYear;
BeginOfTheYear = Januaryl(2006);
```

3.3.8. Об'єднання

Об'єднання – це тип даних, що складається з полів, які фізично відповідають одній і тій самій області пам'яті. Мова Сі має можливість практично так само маніпулювати даними, як і асемблер, об'єднання є однією із таких "низькорівневих" можливостей. Тип даних *union* дозволяє один виділений блок пам'яті інтерпретувати по-різному, що дозволяє зручно маніпулювати даними та одночасно економити пам'ять. За видом опису структура схожа на структуру, наприклад:

```
union
    {
    int a;
    long b;
    char c;
    } myunion;
```

Розмір області пам'яті, що виділяється компілятором, визначається так, щоб помістилося поле максимального розміру. Тобто для вищенаведеного прикладу буде виділено 4 байти для змінної *myunion*. При цьому при зверненні до *myunion.b* буде задіяна вся ця область, при зверненні до *myunion.a* – лише перші 2 байти цієї області, а *myunion.c* дозволить оперувати лише першим байтом. Тобто звернення до однієї й тієї самої області пам'яті як до змінних різного типу.

Об'єднання, як і структура, може використовуватися в описі нового типу, бути елементом масиву і полем структури.

3.4. Вказівники та адреси змінних

Вказівник – це змінна, що містить адресу деякого елемента даних (змінної, константи, функції). У мові Сі вказівники тісно пов'язані з обробкою масивів і рядків.

Для оголошення змінної як вказівника використовується оператор "*":

```
int *p; // p – вказівник на ціле число
```

Для присвоєння адреси деякої змінної вказівнику використовується оператор "&", наприклад:

```
char *p; // оголошуємо вказівник (здатний містити адресу зберігання
будь-якого символу);
char c= 'A';// оголошуємо символьну змінну c;
p = &c; // тепер p містить адресу зберігання символу c.
```

Для того, щоб витягнути значення змінної, на яку вказує вказівник, використовується оператор розіменування "*".

```
char *p;
char b;
char c = 'A';
p = &c;
b= *p; // тепер b дорівнює не адресі зберігання змінної c, а значенню
змінної c, тобто 'A'.
```

Аналогічно, цей оператор можна використовувати і для присвоєння деякого значення змінній, на яку вказує вказівник:

```
char *p;
char b, char c = 'A';
p = &b; // тепер p містить адресу зберігання змінної b, яка ще не
ініціалізована;
*p = c; // ініціалізуємо змінну b значенням 'A'.
```

Стосовно програмування мікроконтролерів, вказівники можна використовувати, приміром, для запису даних в порт введення/виведення. Припустимо, регістр даних порту розташований в пам'яті за адресою 0x16. В цьому випадку, для запису в нього значення 0xFF можна скористатися таким фрагментом програмного коду :

```
unsigned char *p; //оголошення вказівника на символ
p = 0x16; // привласнюємо вказівнику адресу порту
*p = 0xFF; // тепер на виводах порту будуть усі одиниці
```

За допомогою вказівників, використовуваних як параметри функцій, можна організувати повернення більше за одне значення. Розглянемо такий приклад:

```
int SumAndDiv(int *a, int *b) /* Оголошуємо функцію, що повертає суму
від цілочислового ділення без залишку та із залишком */
    {
    int bufA, bufB; // Оголошення цілих чисел - bufA і bufB;
    bufA = *a; // bufA = значенню, на яке вказує вказівник a;
    bufB = *b; // bufB = значенню, на яке вказує вказівник b;
    *a = bufA / bufB; // вказівник a посилається на результат
цілочислового ділення без залишку
    *b = bufA % bufB; // вказівник b посилається на залишок від
цілочислового ділення
    return bufA + bufB; //Функція повертає суму
    }
…
int v1, v2, sum;
v1 = 10;
v2 = 3;
```

```
sum = SumAndDiv(&v1, &v2); //sum=4; v1=3; v2=1
```

У функції *SumAndDiv* спочатку зберігаються у буферних змінних значення, на які вказують вказівники *a* і *b*. Потім в змінну, на яку вказує вказівник *a*, записується результат ділення переданих у функцію значень без залишку, а в змінну, на яку вказується вказівник *b*, – залишок від такого ділення. Оскільки за допомогою вказівників ми безпосередньо зверталися до елементів пам'яті, а не до змінних, то після виходу з функції вміст цих комірок залишається незмінним. При виклику функції *SumAndDiv* в неї передаються адреси змінних *v1* і *v2*, яким у тілі функції відповідають вказівники *a* і *b*.

3.4.1. Вказівники на структури

На структури можна створювати вказівники так само, як і для будь-якого іншого типу даних. Приклад застосування такого вказівника:

```
struct DATE
    {
    int Day;
    int Month;
    int Year;
    }
…
struct DATE MyBirthday, *dateP;
dateP = &MyBirthday; // dateP вказує на структуру MyBirthday
```

В такому випадку для доступу до полів структури через вказівник використовується оператор ->:

```
dateP -> Day=7;
dateP -> Month=8;
dateP -> Year=1974;
```

Можна також використовувати й розіменування вказівника:

```
(*dateP).Day=7;
(*dateP).Month=8;
(*dateP).Year=1974;
```

Указівники також можуть бути полями структури. Це часто використовується для створення в пам’яті ланцюжків однотипних структур, коли кожна попередня вказує на наступну:

```
struct DATE
    {
    int Day;
    int Month;
```

```
    int Year;
    struct DATE *next_date; // вказівник на іншу структуру того ж типу
    }
struct DATE date1, date2;
date1.next_date = &date2;
```

Тепер, до прикладу, оператору *date1.next_date->Year* відповідає доступ до поля *Year* структури *date2* (тобто це еквівалентно запису *date2.Year*).

3.5. Масиви і рядки

Масив – це тип даних, який використовується для представлення послідовності однотипних значень. Масиви оголошуються подібно до звичайних змінних, із вказівкою в квадратних дужках розмірності (кількості елементів у масиві):

```
int digits[10]; //Масив з десяти елементів типу int
char str[10]; //Масив з десяти символів (рядок)
```

Доступ до елементів масиву реалізується за допомогою індексу (порядкового номера елемента, починаючи з 0):

```
digits[0] = 0;
digits[1]=1;
str[0] = 'A';
str[l]='B';
```

Часто набагато зручніше ініціалізувати масив безпосередньо при його оголошенні, наприклад:

```
int digits [10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
char str[10] = {'T', 'h', 'e', ' ', 'v', 'i', 'n', 'e', 'd'};
int n;
char c;
n = digits[2]; //n =2
c = str[1]; //c = 'h'
```

При розробці програм слід враховувати, що компілятор не завжди може відстежити вихід індексу за межі масиву. Іншими словами, якби ми в наведеному вище прикладі використовували оператор:

```
c = str[12];
```

то на етапі компіляції помилки не виникне, але в ході виконання програми змінній *c* буде присвоєно значення "13-го", неіснуючого елемента масиву, тобто взятий вміст області пам'яті, адреса якої на 12 елементів зміщена відносно початкової адреси масиву.

3.5.1. Рядки

Рядок в Сі – це масив типу *char*. Вище був розглянутий приклад оголошення такого масиву, що відповідає рядку "The line". Це оголошення можна було б виконати і за допомогою рядкового літерала:

```
char str[10] = "The line";
```

Проте в такому разі слід пам'ятати, що компілятор неявно завершує рядкові літерали символом '*\0*', і, отже, реальна довжина рядка більше на 1. Це слід враховувати, щоб при ініціалізації масиву не вийти за його межі.

При ініціалізації рядків розмірність масиву можна явно не вказувати:

```
char str[] = "The line";
```

В цьому випадку компілятор визначить розмірність самостійно (Це правило застосовуване і до ініціалізації будь-яких інших масивів).

3.5.2. Багатовимірні масиви

Мова Сі допускає використання багатовимірних масивів, тобто масивів, елементами яких є масиви. Приклади оголошення таких масивів :

```
int a2[10][2]; //Двовимірний масив 10x2
int a3[3][2][5]; //Тривимірний масив 3x2x5
```

Фактично, усі елементи багатовимірного масиву зберігаються в пам'яті послідовно, тому наведені нижче приклади ініціалізації аналогічні:

```
int a[2][3] = { {1, 2, 3},
    {4, 5, 6} };
```

і

```
int a[2] [3] = { 1, 2, 3, 4, 5, 6 };
```

Для доступу до елементів багатовимірного масиву використовуються індекси по кожній розмірності або операції розіменування вказівника. Наприклад, щоб витягти в деяку змінну *n* цифру 4 з представленого вище масиву *a*, можна скористатися одним з двох варіантів:

```
n = a[1][0]; //Витягаємо елемент з "рядка" 1 "стовпця" 0 (перший), тобто - цифру 4
n = *(a+3); //a - вказівник на перший елемент масиву, отже, a + 3 - вказівник на 4-й елемент.
```

3.6. Оператори розгалуджень

Оператори розгалудження використовуються для виконання того або іншого блока коду залежно від деякої умови. До таких операторів в мові Сі належать *if-else* і *switch-case*.

Компілятор може інтерпретувати нуль як "брехню", і не нуль як "істину". Логічні операції можуть об'єднувати кілька умов, що перевіряються. Наприклад:

```
if ((вираз1)&&((вираз2)||(вираз3)))
{
/* Код програми тут виконуватиметься якщо: Вираз "Істина"
(означає не нуль) і хоча б один з виразів 2 і 3 теж "Істина" (означає не
нуль) .*/
};
```

3.6.1. Оператор if-else

У простому випадку оператор *if-else* має таку структуру:

```
if (умовний_вираз)
    {
     блок_коду_1;
    }
    else
    {
     блок__коду_2;
    }
```

Якщо умовний вираз істинний, то виконується блок коду 1, інакше – блок коду 2. При цьому як блок коду 2 допускається використовувати послідовність операторів *else-if*:

```
if (вираз_1)
    {
     блок_коду_1;
    }
    else
    {
    if (вираз_2)
         {
         блок_коду_2;
         }
        else
         {
        if (вираз_3)
             {
             блок_коду_3;
             }
            else
```

```
            {
            блок_коду_4;
            }
        }
    }
```

В даному випадку кожний вираз обчислюватиметься по черзі доти, поки не буде знайдено вираз, що дав істинний результат.

Якщо ж результати обчислень усіх виразів виявляться помилковими, то буде виконаний блок коду 4.

У тих випадках, коли при помилкових результатах усіх умовних виразів не вимагається виконувати жодних дій, можна опустити завершуючий блок коду разом з останньою гілкою *else*. Наприклад:

```
if (a == 1) b=a*3;
    else if (a == 2) b=a+10;
        else if (a == 3) b=0;
```

Якщо вираз в першій строці буде істинним, решта операторів будуть пропущені, в іншому випадку розпочнеться послідовна перевірка виразів, які вказані в наступних рядках, поки не буде знайдено вираз, який дає ненульовий результат, або поки не буде провірено останній вираз. Якщо ні один із виразів не буде істинним, жодні дії зі змінною *b* не буде проведено.

Замість оператора *if-else* можна використовувати умовні вирази. Так, конструкцію вигляду:

```
if (умова) блок_коду_1; else блок_коду_2;
```

можна замінити таким умовним виразом:

```
умова ? блок_коду_1: блок_коду 2;
```

Наприклад:

```
(a == 1) ? b= a*3: b = 0; //Якщо a=1, то b = a*3, інакше b = 0
```

3.6.2. Оператор switch-case

У оператора *if-else* можна використовувати тільки вирази, які зводяться до значення TRUE або FALSE. У тих випадках, коли необхідно застосовувати вирази, що дають довільний числовий результат, зручніше скористатися оператором *switch-case*. Цей оператор дозволяє за допомогою деякої змінної вибирати один з виразів, що відповідає заданій константі. Його синтаксис:

```
switch (вираз)
    {
    case константа-вираз_1 : блок_коду
    case константа-вираз_2 : блок_коду
```

```
case константа-вираз_3 : блок_коду
default :блок_кода
}
```

Наприклад код:

```
if (a == 1) b = a*3; else b = 0; //Якщо a=1, то b = a*3, інакше b = 0
```

можна замінити кодом:

```
switch (a)
  {
  case  1: b = a * 3 ;break;
  case  2: b=a+ 10; break;
  case  3: b= 0; break;
  default;
  }
```

Оператор *break* приводить до негайного виходу з блока *switch*.

Якщо з якихось причин необхідно продовжити перевірку відповідності виразу вибору тим константам, які вказані в гілках *case*, слід прибрати оператори *break*.

Крім того, гілки *case* можна каскадувати. Такий прийом особливо зручний в тих випадках, коли треба, наприклад, перевірити введений символ, не піклуючись про те, чи представляє він велику букву чи малу, або коли треба отримати одну й ту саму реакцію на декілька різних чисел. Розглянемо це на прикладі фрагмента деякої абстрактної програми :

```
switch (input)
  {
  case 'a' : case 'A': DoA(); break;
  case 'b': case 'B': DoB(); break;
  case '0': case '1': case '2': case '3': Do0123(); break;
  case '4': case '5' : case '6': case '7': D04567(); break;
  default: DoDefault(); break;
  }
```

3.7. *Циклічні конструкції*

Циклічні конструкції застосовують для повторення деякого блоку коду на підставі умови циклу. У мові Сі використовуються циклічні конструкції *while*, *for* і *do-while*.

3.7.1. *Конструкція while*

Цикл *while* має такий синтаксис:

```
while (умовний_вираз)
{
 // Виконання тіла циклу, якщо вираз істинний
}
```

Іншими словами, цикли *while* доцільно використовувати в тих випадках, коли відповідний оператор або блок операторів необхідно виконувати доти, поки умовний вираз істиний. Приклад формування рядка, що складається з непарних цифр:

```
int c, i;
const char str[] = "0123456789";
char OddNums[5]; //Рядок для зберігання непарних цифр
c = 0; //Лічильник циклів
i = 0; //Індекс масиву OddNums
while (c < 10) //Доти, поки с менше 10
    {
     //Якщо залишок від ділення з на 2 = 1, то записуємо в і -у позицію
масиву OddNums c-й елемент рядка str, після чого значення і
автоматично інкременується
    if ((c % 2) == 1) OddNums[i++] = str[c];
    c++; //c = c + 1
    }
```

3.7.2. Конструкція for

Цикл *for* має такий синтаксис:

```
for (вираз_1; вираз_2; вираз_3)
{
// Виконання тіла циклу
}
```

Вираз 1 виконується тільки один раз при вході в цикл і зазвичай являє собою оператор присвоювання деякого початкового значення лічильнику циклу. Вираз 2 – це умовний вираз, що визначає момент виходу із циклу (цикл виконується доти, поки він відповідає TRUE або 1). Вираз 3 – ще один оператор присвоєння, в якому зазвичай змінюється лічильник циклу або деяка змінна, що впливає на виконання умови у виразі 2. Вирази можуть бути представлені будь-якими операторами, включаючи порожні (тобто замість виразу можна поставити тільки символ крапки з комою).

Цикли *while* і *for* у більшості випадків взаємозамінні. Так, наведений вище приклад для циклу *while*, можна переписати в такому вигляді:

```
int c, i;
const char str[] = "0123456789";
char OddNums[5]; //Рядок для зберігання непарних цифр
i = 0; //Індекс масиву OddNums
for (c = 0; c < 10; c++) //Доти, поки с менше 10
    if ((c % 2) == 1) OddNums[i++] = str[c];
```

У одних ситуаціях зручніше застосовувати цикли *for*, в інших – *while*. Перевагою циклів *for* є наочніша ініціалізація і організація зміни лічильника циклу. З іншого боку, цикли *while* гнучкіші і забезпечують більше можливостей для реалізації нестандартних програмних рішень при організації обчислень, що повторюються.

3.7.3. Конструкція do-while

Окрім циклів *while* і *for*, в яких спочатку виконується перевірка істинності умови циклу, і тільки потім управління передається блоку операторів циклу, в мові Сі є також конструкція *do-while*. Вона відрізняється від перших двох тим, що в ній спочатку виконується блок операторів, і тільки потім перевіряється виконання умови. Іншими словами, цикл *do-while* завжди виконується як мінімум один раз, незалежно від умови циклу. Цикл *do-while* має такий синтаксис:

```
do
    {
    // Виконання блока операторів циклу
    }
while (умовне_вираження);
```

3.7.4. Організація нескінченних циклів

Для організації нескінченного циклу як умовний вираз у конструкції *while* або *do-while* можна просто вказати значення TRUE або 1:

```
while (1) { блок_операторів; }
```

У разі циклів *for* це виглядатиме так:

```
for (;;) { блок_операторів; }
```

3.7.5. Оператори break і continue

Якщо в тілі будь-якого циклу зустрічається оператор *break*, управління тут же передається на оператор, що йде за оператором циклу, незалежно від істинності або неістинності умовного виразу. При цьому у вкладених циклах вихід здійснюється не на найбільший рівень вкладеності, а лише на один рівень вгору.

При виконанні оператор *continue* усі оператори блока, що знаходяться після нього, пропускаються, а управління передається в початок циклу для наступної ітерації. На практиці оператори *continue* використовуються набагато рідше, ніж оператори *break*. Проте в складних циклах, що вимагають

ухвалення рішень на підставі багатьох чинників, використання операторів *continue* може бути дуже зручним.

3.8. Директиви препроцесора

Директиви препроцесора, по суті, не є складовою частиною мови Сі, а використовуються для підстановки коду в текст програми.

Препроцесор – це особлива програма, яка виконує попередню обробку даних до початку компіляції. Розглянемо стандартні директиви препроцесора.

3.8.1. Директива #include

Директива *#include* використовується фактично в усіх програмах для включення в текст програми заголовних файлів (з визначеннями), що мають розширення файлів *.h* або *.c* (що не містять функцію *main()*). Ця директива може використовуватися в двох формах:

```
#include <ім'я_файлу>
```

чи

```
#include "ім'я_файлу"
```

Якщо ім'я файлу стоїть між знаками "<" та ">", то він вважається частиною стандартної бібліотеки, що поставляється разом з компілятором.

Якщо ж ім'я файлу стоїть між знаками " та ", то вважається, що він розташований в тій самій теці, що і файл з початковим кодом.

3.8.2. Директива #define

Директива *#define* вказує препроцесору на необхідність виконати підстановку, в тексті програми певної послідовності символів іншою послідовністю. Формат директиви:

```
#define замінювана_послідовність фрагмент_підстановки
```

Наприклад:

```
#define MyName "JohnSmith"
#define Condition (a>b)
#define Operation a = b
…
char str[ ] = MyName; //Рівнозначно char str[ ] = "JohnSmith";
int a, b;
if Condition Operatrion; //Рівнозначно if (a>b) a = b;
```

У директиві *#define* можуть використовуватися параметри, завдяки чому вона стає дуже потужним і гнучким інструментом,

що дозволяє замінювати один простий текстовий елемент складними виразами. Такі підстановки називають **макросами**.

Наприклад, вираз, в якому вибирається більше з двох значень, можна представити у вигляді макросу

```
#define larger(x, y) ( (x) > (y) ?(x) : (y) )
```

Якщо визначений такий макрос, то код, що використовує його, може мати такий вигляд:

```
int a = 9;
int b = 7;
int c = 0;
c = larger(a, b);
```

Нагадує виклик функції, проте це не функція – компілятор отримає від препроцесора останній рядок у такому вигляді:

```
c = ( (a) > (b) ? (a) : (b) );
```

Основна відмінність макросу від функції полягає в тому, що код макропідстановки вставляється препроцесором в програмний код скрізь, де зустрічається заданий текстовий елемент. Тоді як код функції визначається тільки в одному місці, а в тих місцях, де вказано її ім'я, здійснюється виклик цього коду.

Існує ще одна відмінність, особливо важлива при програмуванні мікроконтролерів: при використанні макросів, на відміну від функцій, нічого не поміщається в стек, що дозволяє заощадити оперативну пам'ять.

Крім того, макроси перетворяться в звичайний код, який виконується швидше, ніж код функції, на виклик якого і повернення управління у викликувану функцію йде додатковий машинний час. Нарешті, при використанні макросів непотрібне формальне оголошення типів даних.

Перелічимо деякі правила застосування директиви *#define*:

- при створенні коментарів у рядку *#define* завжди використовується коментар вигляду */* … */*;
- слід пам’ятати, що кінець рядка – це кінець *#define*, і весь текст зліва замінить текстом зправа;
- для перетворення параметра макросу в текстовий рядок можна вказати перед ним символ "#", наприклад:

```
#define OutString(s) puts(#s)
OutString(Line); // Рівнозначно puts("Line");
```

- для конкатенації (об’єднання) двох параметрів можна скористатися оператором "##", наприклад:

```
#define Concat(x, y) x ## y
int i = Concat(2, 1); // Рівнозначно int i = 21;
```

- для перенесення тексту підстановки на інший рядок використовується символ оберненої похилої "\", наприклад:

```
#define LongStr "0 1 2 3 4 5 6 7 8 9 10 \
11 12 13 14 15 16 17 18 19 20"
```

- для скасування визначення використовується директива *#undef*, наприклад:

```
#define A_Char 'A'
#undef A_Char
#define A_Char 'a'
```

3.8.3. Директиви умовної компіляції

Багато мікроконтрлерів відрізняються лише деякими параметрами, кількістю виводів, розміром пам’яті і розміщенням регістрів. Директиви умовної компіляції дозволяють створювати мовою Сі програмний код для всього модельного ряду. Однак для цього необхідно якось замінити ті параметри, які відрізняються у різних моделей. Для такої мети застосовуються директиви умовної компіляції: *#ifdef*, *#ifndef*, *#else*, *#endif*, а також *#if* та *#elif*.

Синтаксис для директиви *#ifdef* має такий вигляд:

```
#ifdef ім’я_макросу
    послідовність_операторів_1
#else
    послідовність_операторів_2
#endif
```

Якщо ім’я макросу визначене в програмі, то компілюється перша послідовність операторів, в іншому випадку – друга послідовність (гілка *#else* може бути відсутня).

Синтаксис для директиви *#ifndef* має такий вигляд:

```
#ifndef ім’я_макросу
    послідовність_операторів_1
#else
    послідовність_операторів_2
#endif
```

У цьому разі, на відміну від директиви *#ifdef*, перша послідовність операторів компілюється в тому випадку, коли ім’я макросу в програмі не визначене.

Для умовної компіляції можна скористатися директивами *#if* та *#elif*. Їх синтаксис має такий вигляд:

```
#if вираз_1
    послідовність_операторів_1
#elif вираз_2
    послідовність_операторів_2
#else
    послідовність_операторів_3
#endif
```

Така конструкція працює аналогічно оператору умови *if-else*. Компілятор оцінює вирази після *#if* та *#elif* доти, поки одне з них не дасть результат TRUE, після чого в текст програми підставляється відповідна послідовність операторів. Якщо обидва вирази дають FALSE, то підставляється послідовність операторів після директиви *#else* (якщо вона наявна).

Наведемо приклад застосування директив умовної компіляції. Припустимо, що для передавання і приймання даних у абстрактного мікроконтролера SomeMic16 використовуються виводи 12 і 13, у мікроконтролера SomeMic8 – виводи 6 і 14, у SomeMic4 – виводи 1 і 2. Тоді ми можемо створити заголовний файл SomeMic.h та включити в нього такі директиви:

```
#if SomeMicX == 16
    #define TXD 12
    #define RXD 13
#elif SomeMicX == 8
    #define TXD 6
    #define RXD 14
#elif SomeMicX == 4
    #define TXD 1
    #define RXD 2
#else
    #error "Pins TXD and RXD for SomeMicX are not defined"
#endif
```

Тепер, якщо програмний проєкт буде побудований на мікроконтролері SomeMic8, тоді в заголовку програми слід помістити такий текст:

```
#ifndef SomeMicX
    #define SomeMicX = 8
    #include <SomeMic.h>
#endif
```

Зустрівши директиву *#ifndef*, препроцесор включить у текст програми *#define SomeMic=8* та *#include <SomeMic.h>*. Оскільки

після цього елемент SomeMicX отримає значення, що дорівнює 8, то будь-яке повторне оброблення директиви *#ifndef* не призведе до дублювання у вихідному тексті відповідної інформації. Іншими словами, вміст заголовного файлу SomeMic.h буде поміщено у початковий код файлів, які використовують заголовний файл з *#ifndef* тільки один раз.

Застосування директиви *#ifndef* із наступними директивами *#define* і *#include* – стандартний прийом, який дозволяє уникнути дублювання інформації із заголовних файлів у початковому коді проєкту. Якщо цього не зробити, то при дублюванні компілятор зазвичай видає багато повідомлень про помилки, пов'язані з багаторазовим оголошенням змінних.

3.8.4. Директива #error

Директива *#error* використовується сумісно з директивами умовної компіляції. Зустрівши її, компілятор згенерує повідомлення про помилку, яке вказане зправа від директиви. Наприклад:

```
#error "Pins TXD and RXD are not defined"
```

3.9. Управління регістрами мікроконтролера

Управління роботою мікроконтролера у більшості випадків зводиться до такого простого набору дій із його регістрами:

- запис у регістр необхідного значення;
- читання значення з регістра;
- установка в одиницю потрібних розрядів регістра;
- скидання розрядів регістра в нуль;
- перевірка стану розряду регістра;
- зміна логічного стану розряду регістра на протилежне.

В усіх вказаних діях бере участь оператор присвоєння мови С, записуваний у вигляді знака рівності. Принцип дії оператора зводиться до запису в регістр або змінну, розташовану зліва від нього, значення записаного справа.

```
A = 16; //A присвоїли значення 16
A = B;//A присвоїли значення змінної B
A = B+10;//A присвоїли значення змінної B+10
```

Мова С не має у своєму складі команд безпосереднього скидання або установки розрядів змінної, проте наявні побітові логічні операції "І" та "АБО", які успішно використовуються із цією метою.

Оператор побітової логічної операції "АБО" записується у вигляді вертикальної риски – " | " і може виконуватися між двома змінними, а також між змінною і константою.

Отже, для будь-якого біта логічне "АБО" з логічною одиницею "1" дасть в результаті "1", незалежно від стану цього біта, а логічна операція "АБО" з логічним нулем "0" залишить в результаті стан початкового біта без зміни.

Ця властивість дозволяє використовувати логічну операцію "АБО" для встановлення *n*-го розряду в регістрі.

Для цього необхідно обчислити константу з одиничним *n*-м бітом за формулою 2^n, яка називається бітовою маскою і виконати логічне "АБО" між нею і регістром. Наприклад для встановлення біта №7 в регістрі SREG:

```
SREG = SREG | 128;
```

Таку роботу з регістром прийнято називати "читання-модифікація-запис". На відміну від простого присвоєння, вона зберігає стан інших бітів без зміни.

Наведений програмний код, встановлюючи сьомий біт у регістрі SREG, залишає інші біти регістра без змін.

Єдиний недолік такого запису – це константа 128. По ній нелегко вгадати встановлений сьомий біт, тому частіше маску для *n*-го біта записують у вигляді

```
SREG = SREG | (1<<7);
```

де (1<<N) – вираз мовою Сі, який означає, що число один, зрушено на *n* розрядів вліво. Це і є маска зі встановленим *n*-м бітом. Тоді попередній приклад може бути записаний у вигляді

```
SREG |= (1<<7);
```

Цей код означає: узяти вміст праворуч від знака рівності, виконати між ним і регістром (змінною) ліворуч операцію, що стоїть перед знаком рівності, і записати результат у регістр (змінну) ліворуч.

Ще одна логічна операція мови С – побітове "І", записується у вигляді символу "&". Як відомо, операція логічного "І", стосовно двох бітів дає одиницю тоді і тільки тоді, коли обидва початкових біта мають значення одиниці, це дозволяє застосовувати її для скидання розрядів у регістрах. При цьому використовується бітова маска, в якій усі розряди одиничні, крім біта на позиції того, що скидається. Її легко отримати з

маски зі встановленим *n*-м бітом, застосувавши до неї операцію побітного інвертування "~", яка змінює стан біта на протилежний. Наприклад, (1<<3) – у двійковому вигляді представляє значення у двійковому коді 00001000b. Після інвертування ~(1<<3) отримаємо значення у двійковому коді 11110111b. Отже, скидання сьомого біта в регістрі SREG виглядатиме так:

```
SREG = SREG & (~(1<<7));
```

або

```
SREG &= ~(1<<7);
```

3.10. Література до розділу

1. Brian Kernighan and Dennis Ritchie The C Programming Language, Second Edition .– New Jersey: Prentice Hall, 1988. 232 c.
2. Програмування мікроконтролерів систем автоматики: конспект лекцій для студентів базового напряму 050201 “Системна інженерія” / укл.: А.Г. Павельчак, В.В. Самотий, Ю.В. Яцук – Львів: Львівська політехніка, 2012. 143 с.
3. Програмування мовою Сі для AVR і PIC мікроконтролерів, друге видання. Київ: «МК-Прес», 2012.– 544 с.
4. https://radio-hobby.org/modules/instruction/winavr-i-avr-studio/predisloviye
5. http://microsin.net/programming/pc/c-struct.html

РОЗДІЛ 4. СЕРЕДОВИЩА РОЗРОБЛЕННЯ ПРОГРАМ ДЛЯ МІКРОКОНТРОЛЕРІВ STM32. ВСТАНОВЛЕННЯ, НАЛАШТУВАННЯ, ПОРЯДОК РОБОТИ, СТВОРЕННЯ ПРОЄКТУ

Розроблення застосунку на мікроконтролері не завершується використанням мікроконтролера в схемі з необхідним "обвісом", потрібна також програма згідно з якою він буде працювати, щоби пристрій виконував задумані розробником функції.

Програма для мікроконтролера пишеться на однією з мов програмування у вигляді текстового файлу. Це означає, що для написання програми можна скористатися будь-яким текстовим редактором. Спеціальна програма-транслятор перетворює початковий текст програми в машинні коди, зрозумілі мікропроцесору, цей процес називається компіляцією. В результаті компіляції буде створений так званий hex-файл, призначений для завантаження в пам'ять мікроконтролера. Якщо відкрити hex-файл звичайним блокнотом, то він складатиметься з рядків шістнадцяткових цифр. Тому файл і називають "hex" – від англійського "hexadecimal" – "шістнадцяткова система числення".

Для полегшення процесу розробки програми використовується інтегроване середовище програмування, до складу якого включається певний набір програмних засобів: редактор початкового тексту, транслятори з вибраної мови програмування, редактори зв'язків, завантажувачі, відлагоджувачі і так далі. При розробці програми, крім файлу з самим текстом програми, середовище розробки використовує безліч допоміжних файлів. Для того, щоб усі ці файли були належно взаємозв'язані, створюється програмний проєкт. Отже, для написання й налагодження програми існують різноманітні засоби.

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.

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

- записувати код і переміщуватися 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 потрібні:

- 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).

4.1. Середовища розроблення (IDE, Integrated Development Environment) і компілятори

4.1.1. IDE

Під ARM-архітектуру існує досить широкий вибір програмних засобів розробки. Наведемо найпопулярніші програмні пакети:

- **Keil uVision (MDK-ARM)**. Keil постачає свою IDE та свій компілятор Keil C/C++. Містить надлагоджувач, симулятор і ряд корисних бібліотек та інструментів. Працює з програматором-надлагоджувачем ULink, ST-

Link. Існує безкоштовна версія пакета «Lite» із обмеженням на величину коду в 32 Кб.

- **ARM Development Studio** (ARM купила Keil). ARM DS базується на Eclipse, може працювати як зі своїм компілятором, так і з GCC. Містить налагоджувач, симулятор і ряд корисних бібліотек та інструментів. Працює з програматором-відлагоджувачем ULink, ST-Link.
- **Embedded Workbench for ARM (EWARM)** від IAR Systems. Власний IDE та компілятор IAR C/C++. Працює з програматором-налагоджувачем J-Link, ST-Link. Існує безкоштовна версія із обмеженням коду в 32 Кб або повним функціоналом на 30 днів.
- **TrueSTUDIO for ARM** від Atollic (на сьогодні куплена компанією STMicroelectronics) базується на Eclipse і включає компілятор GCC C/C++. Працює з програматором-відлагоджувачем ST-Link, STICE. Існує безкоштовна версія, із тим же обмеженням в 32Кб (8Кб для Cortex-M0 і M1), за функціоналом обмеження не має.
- **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.
- **Ride7+RKIT-ARM** від Raisonance, який включає компілятор GCC C/C++. Працює з програматором-налагоджувачем RLink. Не має обмежень за налагодженням.
- **CoIDE** від CooCox дуже хороша, особливо для початківців, оболонка. Недоліком є відсутність нормальної підтримки C++, відсутність версії під Linux.

Однак, як зрозуміло з написаного вище, для ARM/ARM Cortex існує підтримка в gcc, який, у свою чергу, «дружить» із Eclipse. тому наявні відкриті, Open Source, засоби:

- **GNU Tools for ARM Embedded Processors** відкритий (Open source) компілятор GCC;

- **Eclipse** відкрита (Open source) IDE. Включає компілятор GCC C/C++. Працює з програматором-налагоджувачем ARM-Link. Не має обмежень.

Маючи трішки часу, досвіду та бажання попрацювати, гарне середовище можна зібрати самостійно із ARM-GCC та Eclipse.

4.1.2. Допоміжні засоби

Крім компілятора, оболонки та засобів для прошивки і відлагодження, існує ряд корисних програм, які суттєво полегшують розроблення застосунку:

- **STM Studio run-time variables monitoring and visualization tool**, від STMicroelectronics. Інструмент, за допомогою якого можна стежити за змінними у мікроконтролері під час його роботи. Працює з використанням стандартних засобів відлагоження, типу ST-Link. Безкоштовна.
- **MicroXplorer, MCU graphical configuration tool** та **STM32CubeMX** від STMicroelectronics. Засоби конфігурування (та підбору оптимальних комбінацій) пінів мікроконтролерів, в залежності від моделі та периферії, із якою планується працювати. Враховуючи кількість периферії та велику кількість ніжок мікроконтролерів, є зручними засобами.
  STM32CubeMX – кодогенератор з графічним інтерфейсом, який максимально спрощує налаштування мікроконтролерів STM32, який дозволяє автоматично генерувати конфігураційний С-код і "скелет" прикладної програми за допомогою набору візуальних утиліт.
- **CooCox CoSmart** - засіб конфігурування (та підбору оптимальних комбінацій) пінів мікроконтролерів, в залежності від моделі та периферії, із якою планується працювати.
- **CooCox CoAssistant** інтерактивний довідник по регістрам. Можна користуватися он-лайн.

4.1.3. Операційні системи для STM32

Навіть для мікроконтролерів є ніша для операційних систем. Нижче наведено короткий список (деяких з них) сумісних із STM32:

- **ChibiOS/RT**. Портабельна, компактна, багатозадачна OS реального часу. Підтримує ряд сімейств мікроконтролерів з архітектурами ARM7/ARM Cortex-Mx/STM8/MSP430/AVR та інші. Підтримує багатозадачність із витісненням, засоби роботи з багатопоточністю, статична за природою (що важливо для мікроконтролерів), але може працювати і динамічно виділяючи ресурси, Hardware Abstraction Layer (HAL) для великого набору периферії (для STM32F1xx підтримується найбільший набір периферії). Open Source – GPL, GPL with Linking Exception. Є можливість покупки комерційної ліцензії.
- **CooCox CoOS**. Малого розміру (мінімальне ядро – 974 байт), багатозадачна OS реального часу для ARM Cortex M. Підтримує багатозадачність із витісненням, засоби роботи з багатопоточністю, вміє розпізнавати переповнення стеку. Open Source – BSD.
- **Free RTOS**. Дуже популярна і відома. Портабельна, підтримує три десятки архітектур, багатозадачна ОС реального часу. Підтримує як кооперативну багатозадачність, так і багатозадачність із витісненням, а також гібридні варіанти. Засоби для роботи з багатозадачністю, програмні таймери, розпізнавання переповнення стеку. Ліцензія – GPL with Linking Exception, але включає "апаратуро-специфічні" файли зі своїми ліцензіями. Існує декілька модифікацій – OpenRTOS для комерційного використання та SafeRTOS, підвищеної надійності.

4.1.4. Загальні висновки та зауваження

Найбільш популярними (але і найдорожчими) серед розробників для розробки ПЗ під ARM архітектуру являється інструментарії від компаній Keil і IAR Systems. Це зобумовлено найбільш просунутими С-інструментаріями з точки зору оптимізації і компактності коду. Крім лідируючих позицій в С-інструментаріях, ці компанії надають широкі набори додаткового ПЗ – операційні системи реального часу, USB-стеки, TCP/IP-стеки і багато іншого, але за додаткову плату. До того ж компанія Keil належить ARM, і при користуванні

послугами цих двох компаній ви отримуєте дуже добру технічну підтримку.

Підкреслимо популярність засобів на основі компілятора GCC. Існують як платні їх варіанти, так і безкоштовні. Окрім усього, GCC є лідером з кількості підтримуваних процесорів і операційних систем. Приклад варіанта платних засобів-інструментарії від компаній Raisonance і Atollic. В порівнянні з двома раніше описаними варіантами ви отримуєте за набагато менші гроші повноцінний C – інструментарій із середовищем розробки і технічною підтримкою. Існує також варіант повністю безкоштовного інструментарію, наприклад, середовище розробки Eclipse і компілятор GCC.

Важливе зауваження – різні оболонки по-різному генерують допоміжний код ініціалізації, який виконуватиметься до main. Тому можуть бути сюрпризи – наприклад, одні оболонки вмикатимуть за замовчуванням якусь периферію, інші – ні, тоді та ж програма буде успішно працювати в перших і не працюватиме в інших.

Слід пам'ятати, що типовий у світі мікроконтролерів спосіб підключити бібліотеки – це скопіювати їх у папку проєкту, і компілювати разом із власним кодом. Це простіше, ніж тягнути із середовищем безліч комбінацій скомпільованих бібліотек, а сумарний їх розмір невеликий, через обмежену пам'ять контролера. Одні оболонки такий процес автоматизують (як уже згадана CoIDE), інші ні (і тоді в інструкціях пише щось типу "скопіюйте cmsis.c і cmsis.h у папку вашого проєкту").

В Інтернеті можна знайти дуже багато корисних бібліотек (різної якості та з різними ліцензіями) для ARM Cortex взагалі та STM32 зокрема. Важливо, що, якщо потрібно терміново працювати з файловою системою FAT чи дисплеєм на контролері HD44780, ймовірно, писати їх підтримку самостійно або принаймні писати її з нуля не доведеться.

Крім того, можна спробувати програмувати контролер за допомогою Lua (проєкт eLua) або Scheme (Armpit Scheme).

4.2. Keil uVision

Це програмне середовище є інструментарієм від IAR Systems. Характеризується універсальністю, підтримує більшість відомої

архітектури мікроконтролерів таких виробників, як STMicroelectronics, Texas Instruments, Microchip, Atmel і та ін.

Завантажуємо Keil з офіційного сайту: http://www.keil.com/. Тут необхідно викачати і встановити MDK-ARM (потрібна реєстрація на сайті розробника).

Для підтримки мікроконтролерів серії певної фірми необхідно викачати і встановити останню версію бібліотеки стандартної периферії від фірми розробника «xxxxxxxx.pack».

Під час інсталяції Keil має встановитися драйвер для St-Link програматора. Може вимагатися встановити драйвери JTAG-налагоджувача Segger Jlink.

Після встановлення запускаємо Keil і кнопкою на панелі інструментів відкриваємо "PackInstaller".

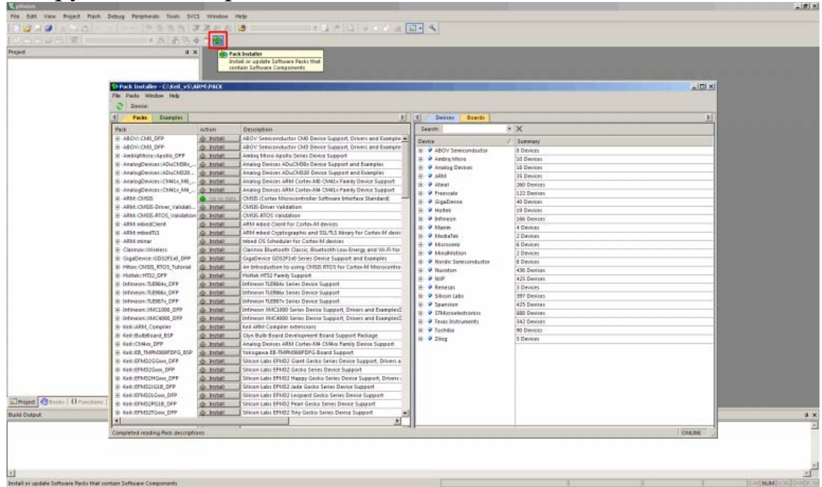


Рис. 4.1. "PackInstaller"

Праворуч на вкладці "Devices" знаходимо цільовий контролер, після чого у вкладці "Packs" зліва по черзі інсталюємо всі пакети.

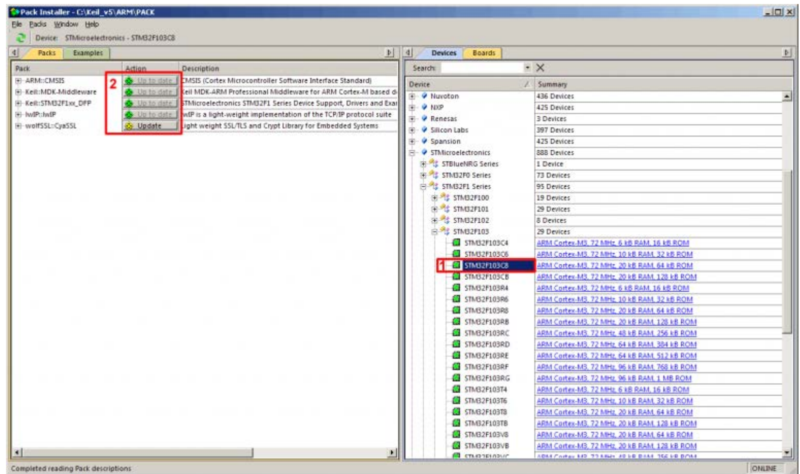


Рис. 4.2. Інсталювання пакетів

Після того, як встановимо всі пакети, закриваємо "PackInstaller".

Після установки всіх компонентів середовища можна приступати до створення першого проєкту. Для цього необхідно створити новий проєкт за допомогою меню "Project -> New mVision Project..." і вибрати директорію для нового проєкту (у назві шляху до директорії не повинно бути кириличних символів).

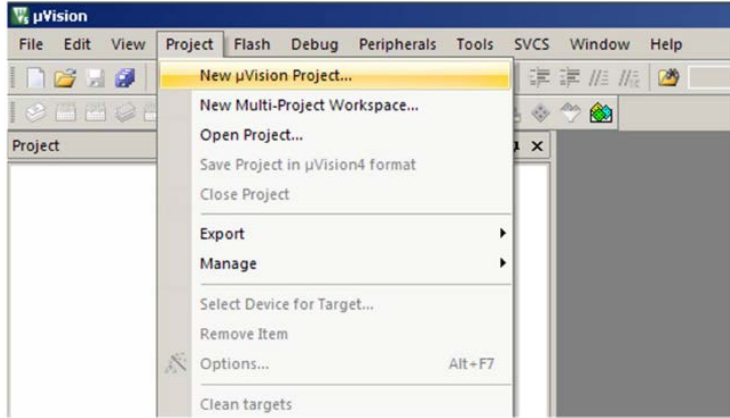


Рис. 4.3. Створення нового проєкту

Вибираємо папку та ім'я проєкту.

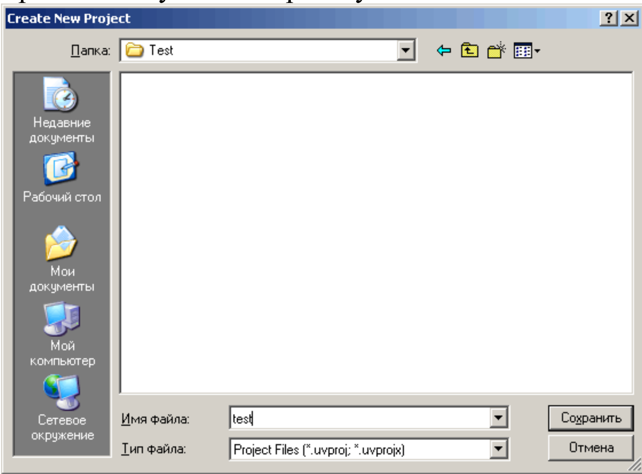


Рис. 4.4. Вибір папки та імуні проєкту

Потім буде запропоновано вибрати цільовий пристрій для якого буде написана програма. Вибираємо мікроконтролер, з яким будемо працювати.

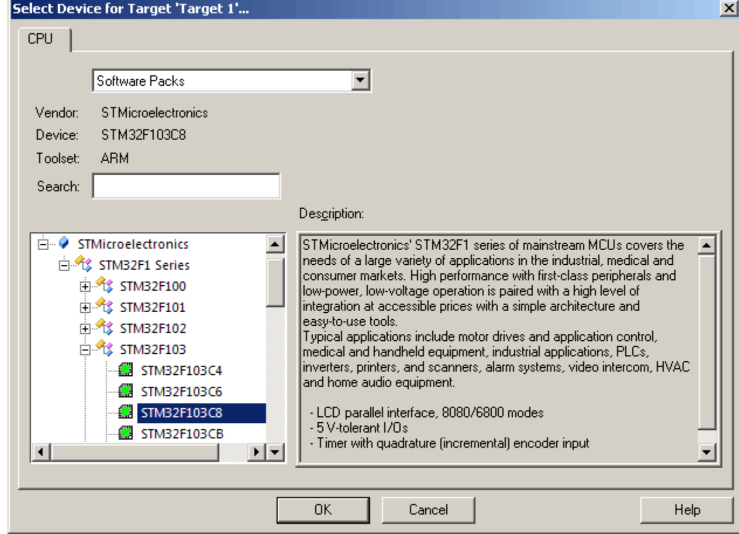


Рис. 4.5. Вибір мікроконтролера

Після вибору цільового мікроконтролера необхідно визначити периферію й оточення компіляції (компоненти які будуть використовуватися у проєкті). Для першого проєкту виберемо Device-ядро застосовуваних мікроконтролерів (підтримка ядра мікроконтролера даної серії), Drivers (порти введення/виведення), Drivers_CLK (сигнали тактової частоти).

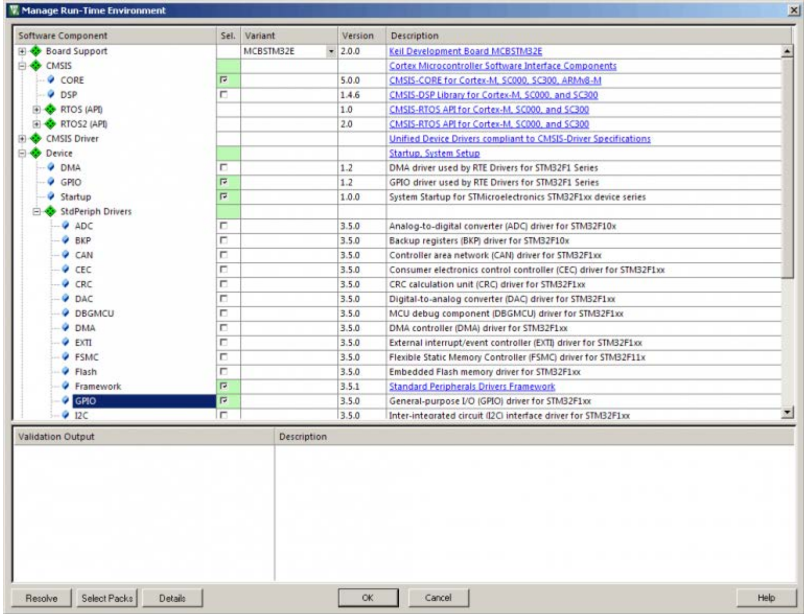


Рис. 4.6. Вікно вибору оточення компіляції

Обов'язково обрати:

- CMSIS-CORE – підтримка основного ядра ARM;
- System Startup – основний конфігураційний системний файл;
- Standard Peripherals Drivers Framework – стандартні драйвери периферії.

Компоненти, що будуть використовуватись у проєкті:

- GPIO – управління ногами мікроконтролера;
- RCC – управління тактуванням периферії;
- та ін.

Після того, як розставили галочки натисніть у лівому нижньому кутку кнопку "Resolve" і тільки потім "OK". Вікно має закритися.

Перед тим як набрати та компілювати програму, потрібно зробити деякі налаштування проєкту. Натискаємо на "Options for Target ..." (ALT+F7) або в дереві проєкту клацаємо правою кнопкою миші на "Target 1".

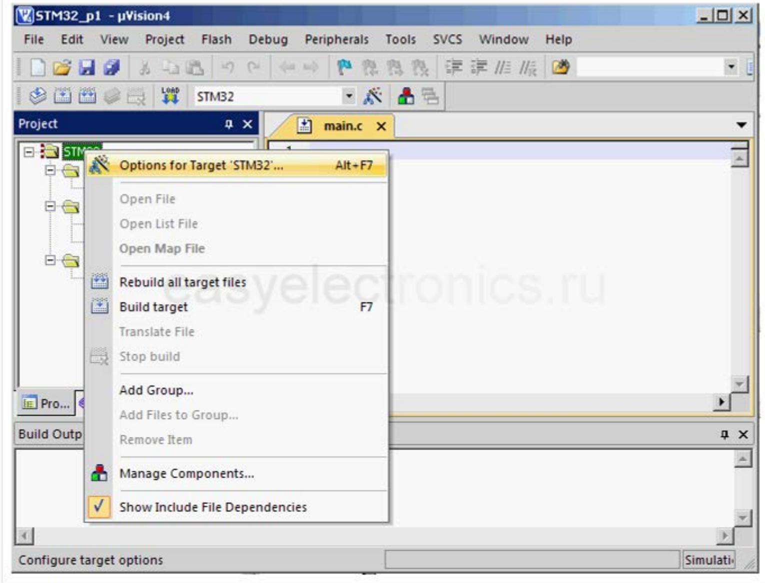


Рис. 4.7. Вікно налаштувань проєкту з призначенням вкладок

Відкриється вікно налаштувань проєкту з призначенням вкладок:

- Target – налаштування параметрів компіляції та емуляції (зокрема, можна налаштувати тактову частоту емуляції);
- Output – генеровані вихідні файли компіляції (наприклад, ставимо прапорець Create Hex file, інакше *.hex-файл не буде згенеровано для подальшого завантаження в мікроконтролер);
- Listing – налаштування текстових лістингів;
- User – запуск різноманітних розширень після компіляції;

- C/C++ – різноманітні налаштування C-компілятора, зокрема оптимізація або за розміром коду, або швидкодією;
- Debug – вибір налагоджувача. Вибравши «Simulator» проводиться симуляція виконання коду програми на комп'ютері. Вибравши програматор, можете "прошиванти" код програми в мікроконтролер і здійснювати налагодження програми в пристрої. Переконайтеся, що встановлений прапорець "Run to main()", щоби налагодження запускалося з функції main().

На вкладці "Target" вікна налаштувань потрібно вказати тактову частоту мікроконтролера.

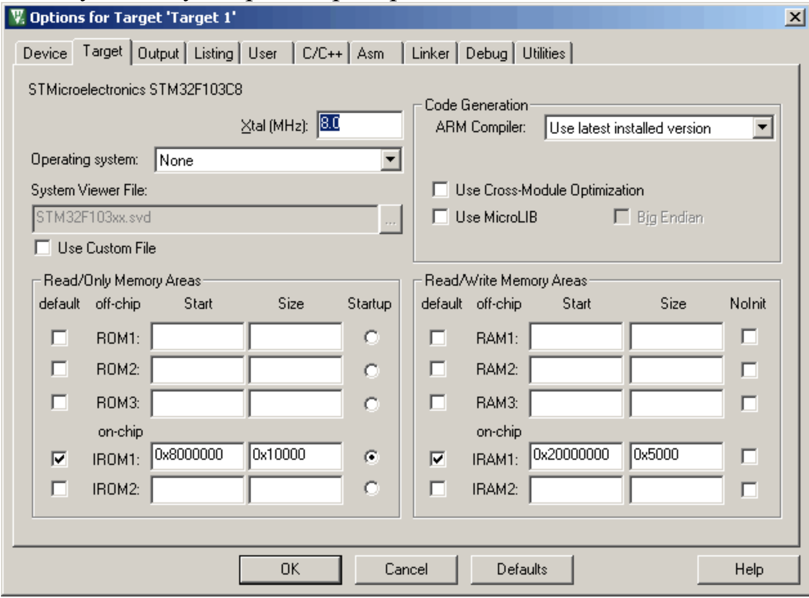


Рис. 4.8. Вікно налаштувань Target 1, вкладка "Target"

На вкладці "Output" вказуємо формат вихідного файлу.

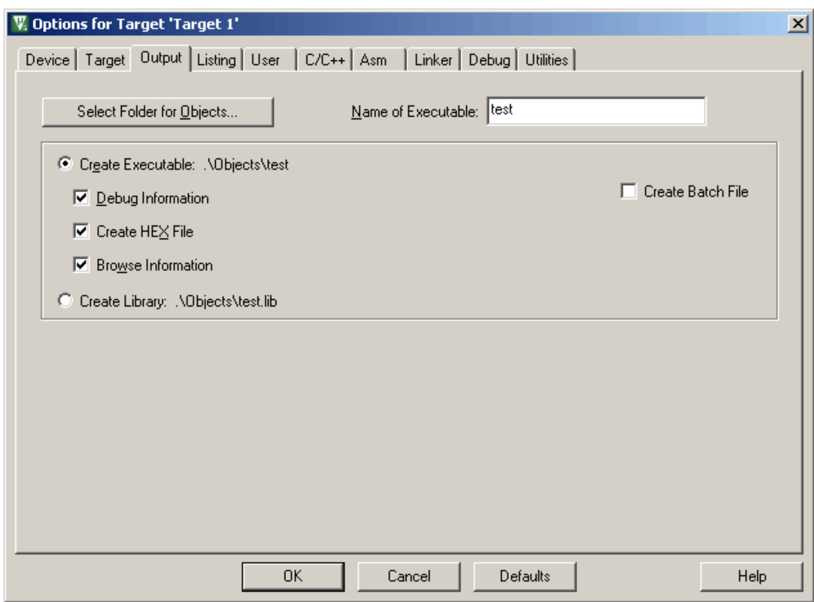


Рис. 4.9. Вікно налаштувань Target 1, вкладка Output

На вкладці "C/C++" у полі "Define:" вказуємо такі опції:
USE_STDPERIPH_DRIVER, STM32F10X_CL

Без цих опцій проєкт не буде нормально компілюватися. Можна вибрати рівень оптимізації. Можливо, потрібно буде вказати шляхи до заголовних файлів CMSIS. В STM32 вони розкладені по двох папках: "CMSIS\CoreSupport\" і "CMSIS\DeviceSupport\ST\STM32F10x\".

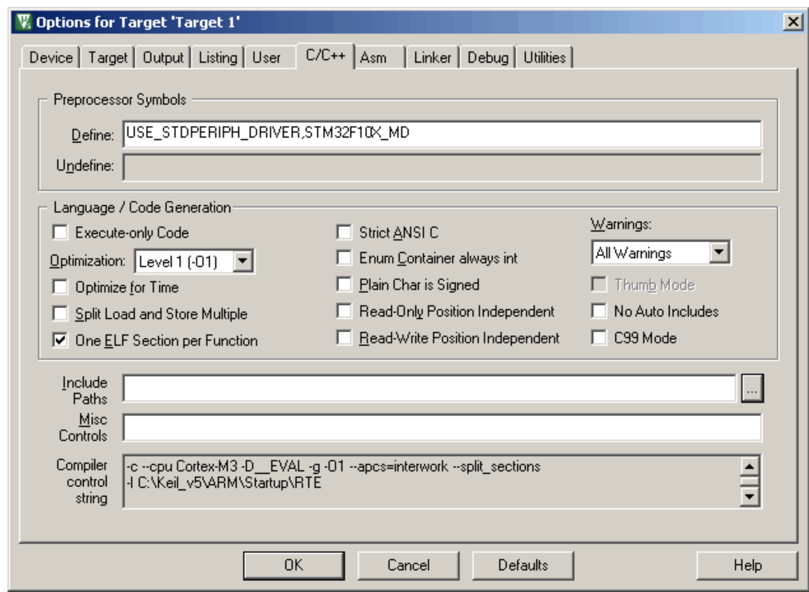


Рис. 4.10. Вікно налаштувань Target 1, вкладка C/C++

Якщо готовий проєкт звідкись переносимо, то шляхи до заголовних файлів теж потрібно тут прописати, інакше компілятор їх не знайде.

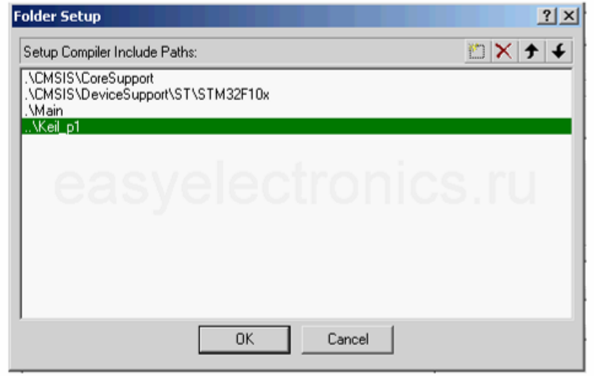


Рис. 4.11. Вікно шляхів доступу до заголовних файлів

Завершальний крок – налаштування JTAG-налагоджувача. Підключивши програматор або налагоджувальну плату до комп'ютера за допомогою кабелю USB, упевнитися, що операційна система правильно виявила і встановила драйвери пристрою.

Для налаштування необхідно перейти на вкладку "Debug".

Рис. 4.12. Вікно налаштувань Target 1, вкладка Debug

У випадному списку "Use:" вибрати налагоджувач, наприклад це може бути "J-Link". Потім натиснути кнопку "Settings". У випадному меню "Port" необхідно вибрати режим "SW" і задати швидкість передачі даних "Max Clock" 2 MHz. Упевнитися, що в полі "SW Device" наявне налаштування ARMCoreSightSW-DP.

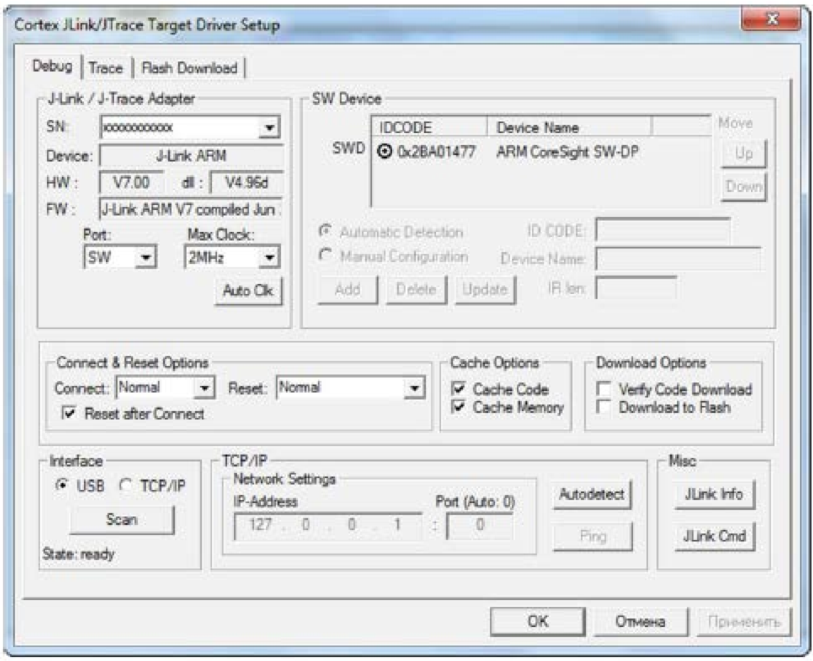


Рис. 4.13. Вікно налаштувань відладчика, вкладка Debug

Далі необхідно задати порядок завантаження програми в пам'ять мікроконтролера. Для цього перейти на вкладку "Flash Download", натиснути кнопку "add" і зі списку вибрати "1986ВЕ IAP 128kB Flash". Потім можна відмітити позиції: "Erase Full Chip", "Program", "Verify", "Reset and Run" на вкладці "Flash Download".

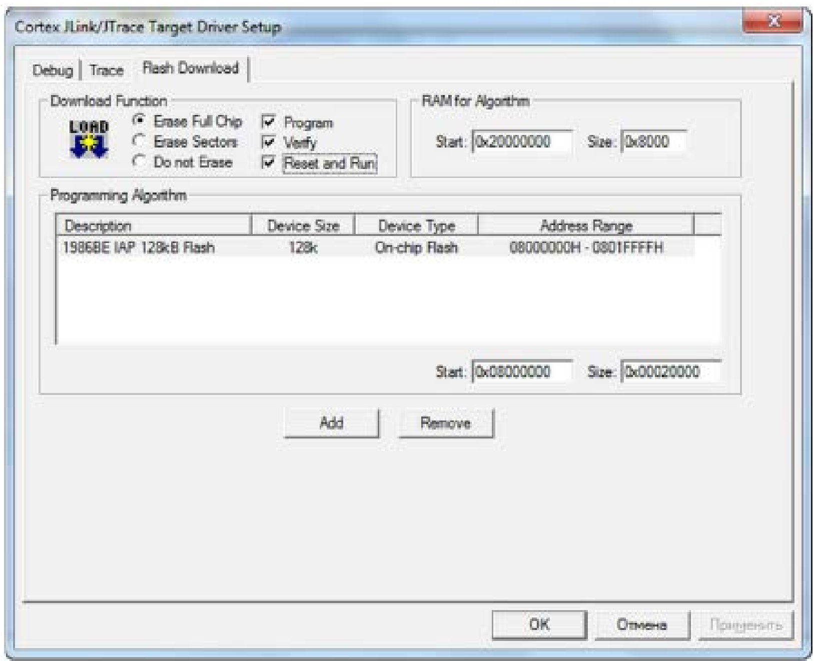


Рис. 4.14. Вікно налаштувань налагоджувача, вкладка Flash Download

Тепер можна закрити вікно налаштувань, проєкт налагоджений і готовий до роботи: можна вводити текст програми, налагоджувати її та компілювати.

Створюємо основний файл програми. Для цього необхідно додати в проєкт новий файл "main.c". В дереві проєкту клацнути правою кнопкою миші на "Sourcegroup 1", далі "Add New Irem to Group Source Group 1...".

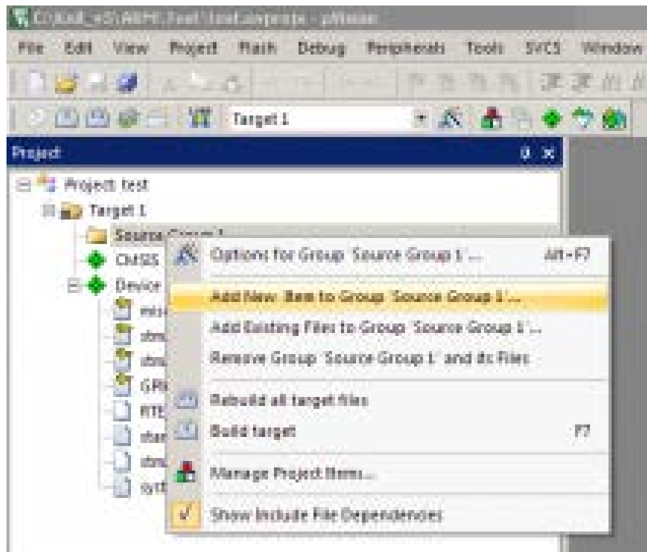


Рис. 4.15. Створення основного файлу програми

У вікні, що з'явилося, вибрати тип файлу "CFile (.c)", задати йому ім'я "хххххх.с" і зберегти в теку в тій самій директорії, що й проєкт.

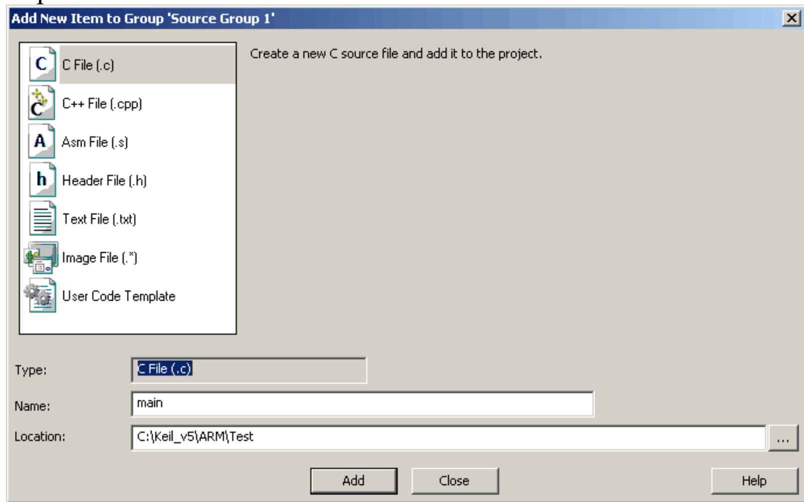


Рис. 4.16. Тип та ім’я файлу

Відтак буде створено й відкрито файл main.c. У вікні, що з'явилося, можна писати код програми. Наберіть в ньому свій код програми, наприклад:

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"

int main(void)
{
 int i;
 /* Initialize Leds mounted on STM32 board */
 GPIO_InitTypeDef GPIO_InitStructure;
 /* Initialize LED which connected to PC13, Enable the Clock*/
 RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
 /* Configure the GPIO_LED pin */
 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
 GPIO_Init(GPIOC, &GPIO_InitStructure);
 while (1)
     {
     /* Toggle LED which connected to PC13*/
     GPIOC->ODR ^= GPIO_Pin_13;
     /* delay */
     for(i=0;i<0x100000;i++);
     }
}
```

або таку

```
#include "stm32f10x.h"

void InitAll(void)
{
 RCC->APB2ENR     |= RCC_APB2ENR_IOPAEN; // Подаємо тактування
на порт, інакше не працюватиме
 GPIOA->CRL     &= ~GPIO_CRL_CNF3;     // Обнуляємо біти CNF3,
оскільки після старту вони налаштовані як режим Open Drain (01), а
нам потрібно Push-Pull (00)
 GPIOA->CRL |= GPIO_CRL_MODE3_0; // Налаштувати порт на вихід
(GPIOA-CRL-MODE після зкидання рівні (00), але в реальній програмі ці
моменти потрібно перевіряти і робити повну ініціалізацію порту
 return;
}

int main(void)
{
```

```
InitAll();
while(1)
    {
    GPIOA->BSRR =GPIO_BSRR_BS3;       // Виставили біт 3
GPIO_BSRR_BS3 це бітмаска
    GPIOA->BSRR =GPIO_BSRR_BR3; // Зкинули біт 3
    }
}
```

Рис. 4.17. Головне вікно програми Keil uVision5

Далі треба скомпілювати код програми. Для цього необхідно вибрати меню "Project -> Build target" або натиснути на відповідну кнопку на панелі інструментів або "гарячою" клавішею F7.

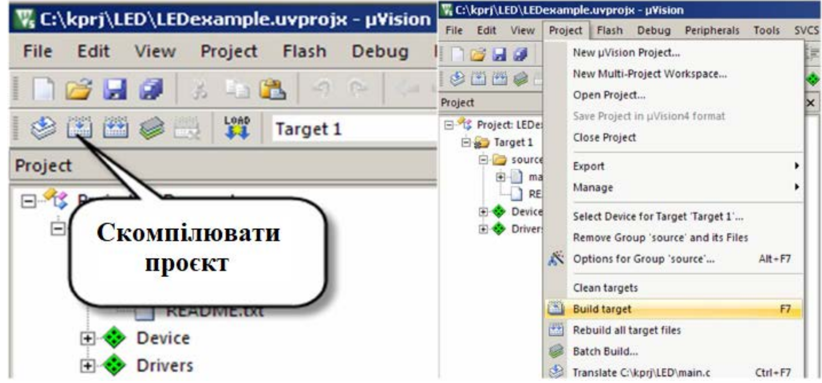


Рис. 4.18. Компіляція коду програми

Скомпільована без помилок програма може бути завантажена в пам'ять мікроконтролера. Для цього треба вибрати "Flash-Download" або натиснути на відповідну кнопку на панелі інструментів.

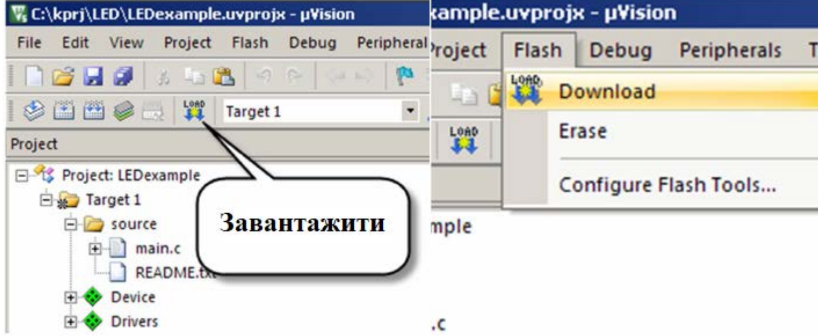


Рис. 4.19. Завантаження скомпільованого коду програми в пам'ять мікроконтролера

Для налагодження програми в середовищі програмування Keil uVision можуть бути використані засоби JTAG-налагоджувача. Для цього необхідно натиснути на кнопку «start/stop debug». Режим налагодження надає можливість ставити точки зупинки виконання програми на мікроконтролері.

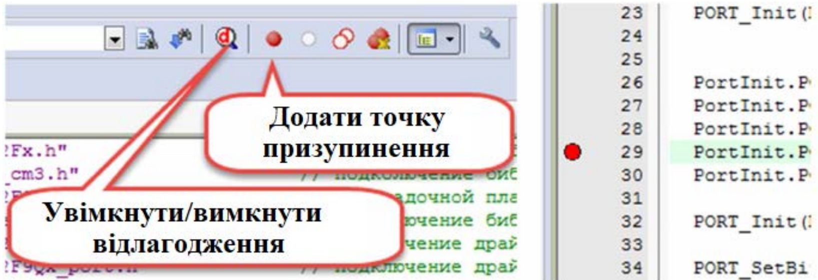


Рис. 4.20. Використання режиму налагодження

Після натиснення кнопки "start/stop debug" вікно програми Keil uVision змінюється. При цьому програма на мікроконтролері не виконується доти, поки користувач не натисне відповідну кнопку.

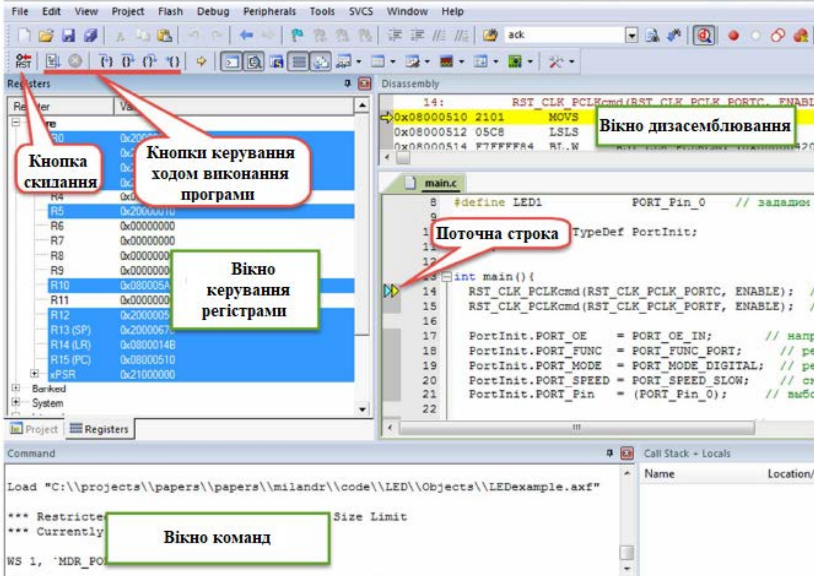


Рис. 4.21. Головне вікно в режимі налагодження

Для управління ходом виконання програми передбачений відповідний блок кнопок:

- Кнопка "Run" запускає виконання програми на мікроконтролері;

- Кнопка "Stop" зупиняє виконання програми;
- Кнопка "Step" виконує перехід до наступного рядка по ходу виконання програми;
- Кнопка "Step over" виконує перехід до наступного рядка по ходу виконання програми, не заходячи в поточну функцію;
- Кнопка "Step out" виконує перехід до точки виходу з поточної функції.

У будь-який момент часу поточний рядок підсвічується за допомогою символу стрілки у вікні відображення коду програми. Над вікном коду програми відображається вікно дизасемблювання, в якому показується хід виконання програми в кодах асемблера. Зліва від вікна коду програми відображається вікно управління регістрами мікроконтролера, що дозволяє переглядати і редагувати всі регістри мікроконтролера. Під вікном управління регістрами розташовується командне вікно, що дозволяє виконувати довільні команди у будь-який момент часу.

Задля спостереження за станом портів відкриваємо вкладку периферії і вивести вікно порта IO "Peripherials -> General Purpose IO -> GPIOA".

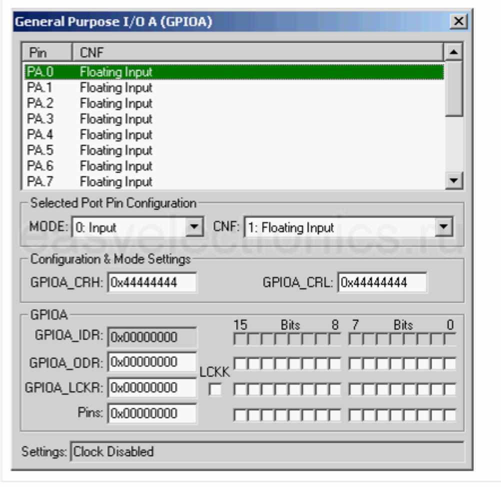


Рис. 4.22. Вікно порта IO

У ньому відразу видно всі режими, в якому знаходиться порт, а також регістри конфігурації. Виконуючи покроково програму можна спостерігати за вмістом порту.

Для виходу з режиму налагодження необхідно натиснути на кнопку "start/stop debug".

4.3. Програма початкових налаштувань STM32CubeMX

STM32CubeMX – графічний редактор для конфігурації мікроконтролерів STM32, що дозволяє автоматично генерувати конфігураційний С-код і «скелет» прикладної програми за допомогою набору візуальних утиліт. Завдяки STM32CubeMX етап створення функцій ініціалізації відбувається дуже швидко. При цьому від програміста не потрібно написання ні єдиного рядка коду, хоча, звичайно, алгоритмічну і смислову частині програми як і раніше треба створювати вручну.

STM32CubeMX дозволяє:

- створювати та редагувати проєкти для мікроконтролерів STM32 з наступною генерацією С-коду для конкретних інтегрованих середовищ: IAR™ EWARM, Keil™ MDK-ARM, Atollic® TrueSTUDIO та AC6 System Workbench (SW4STM32);
- виконувати пошук оптимального мікроконтролера або налагоджувальної плати з урахуванням особливостей конкретної програми. Фільтрування проводиться за різними полями, як і у випадку зі ST MCU Finder;
- проводити візуалізоване налаштування виводів мікроконтролера за допомогою утиліти Pin Wizard з можливістю формування табличного файлу CSV для трасувальників друкованих плат;
- проводити візуалізоване налаштування дерева тактування за допомогою утиліти Clock Wizard, у тому числі – глобальних тактових сигналів та тактових сигналів периферійних пристроїв;
- проводити візуалізоване налаштування периферійних блоків та програмного забезпечення (ПЗ) проміжного рівня, файлової системи, стеків протоколів, операційної системи тощо за допомогою утиліти "Perepherial and

middleware Wizard" (вкладка "Configuration"). При цьому для створення С-коду можуть використовуватися різні бібліотеки від ST: HAL або LL;
- проводити оцінку рівня споживання та термін служби акумулятора при заданих налаштуваннях мікроконтролера за допомогою утиліти "Power consumption Wizard";
- виконувати автоматичне оновлення як самого середовища STM32CubeMX, так і програмних бібліотек STM32Cube, що нею використовуються, для різних сімейств STM32;
- автоматизувати процес міграції проєктів під час переходу від однієї моделі мікроконтролера до іншої.

Програма STM32CubeMX значно полегшує роботу програмістів, проте її можливості не безмежні. Варто відзначити існуючі обмеження:
- генерований С-код охоплює тільки налаштування блоків і периферії мікроконтролера. Це означає, що алгоритмічну частину програми автоматично згенерувати не можна, її потрібно буде дописати вручну;
- STM32CubeMX допоможе створити лише стартову конфігурацію. Іноді в процесі роботи користувачеві потрібно змінити частоту роботи периферійного блока або конфігурацію виведення. Усе це доведеться прописувати самостійно;
- для генерації коду використовуються стандартні, розроблені ST, бібліотеки нижнього рівня (HAL та LL) та проміжного рівня, наприклад, StemWin або STM32_USB_Device_Library;
- в процесі генерації С-файл вибудовується так, що для користувача виділяються спеціальні секції, в які він може поміщати свій код. Якщо код користувача виявиться поза цими рамками – він буде затертий при наступних генераціях;
- існують інші обмеження для окремих блоків, для більш детального ознайомлення з якими слід звернутися до посібника з STM32CubeMX.

Отже, STM32CubeMX – базовий інструмент для створення коду ініціалізації, особливо корисний для новачків у програмуванні на платформі STM32. Це програмне забезпечення вільно поширюється і є частиною ініціативи STCube, мета якої – надання розробникам повного набору інструментів і бібліотек для прискорення процесу розробки. 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 проєкту (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.

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

- керування ST HAL для вибраного сімейства MCU (CubeF0, CubeF1 тощо);
- додаткові функції бібліотеки програмного забезпечення, необхідні для проєкту (бібліотека FatFs, FreeRTOS тощо);
- середовище розробки, яке буде використовуватися для створення програми (IAR, TrueSTUDIO тощо).

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

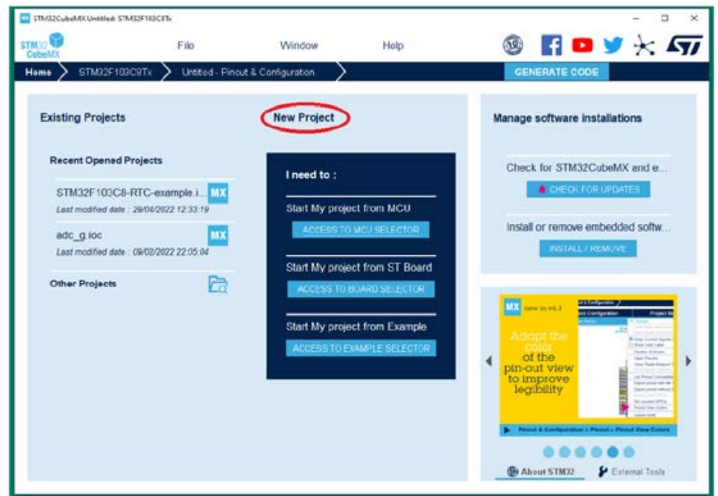


Рис. 4.23. Головне вікно програми STM32CubeMX

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

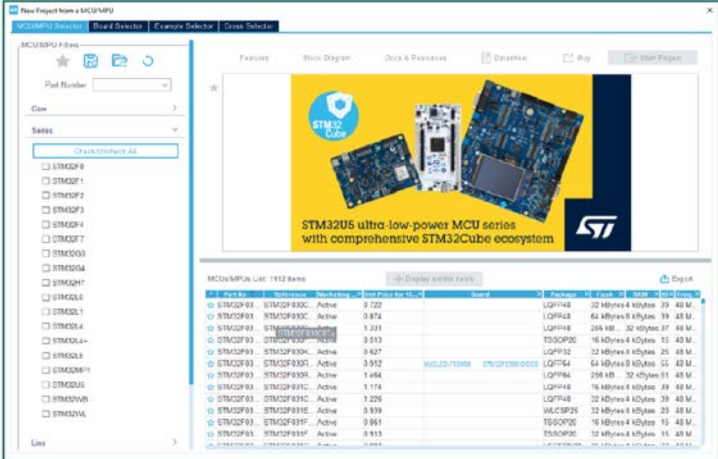


Рис. 4.24. Вікно нового проєкту з вибором типу мікроконтролера

Вікно нового проєкту 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.

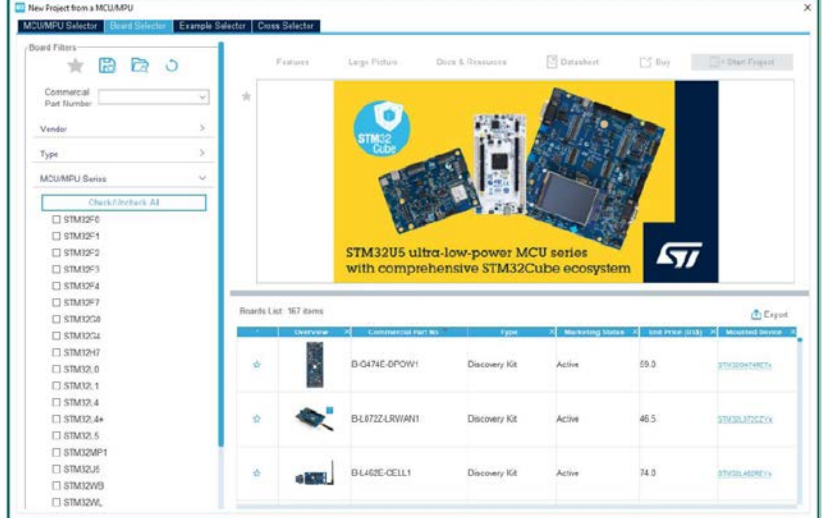


Рис. 4.25. Вибір плати розробки STM32

Вкладка "Board Selector" дозволяє фільтрувати серед усіх офіційних плат розробки ST. На вибір 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.

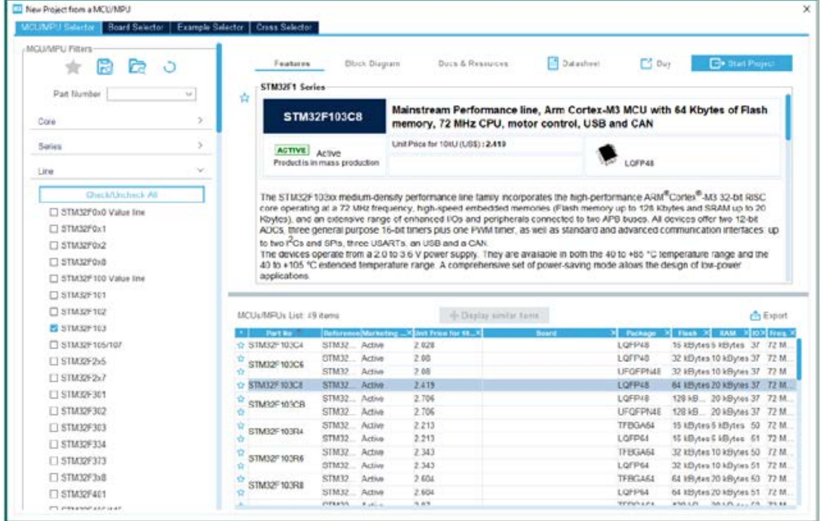


Рис. 4.26. Результат вибору потрібного MCU в STM32CubeMX

Після того, як потрібний мікроконтролер вибрано, у вікні нового проєкту з’явиться панель з короткими характеристиками мікроконтролера та меню вибору додаткової інформації по ньому. Можна подивитися структуру мікроконтролера, отримати додаткову літературу, необхідну для розробки, та довідатися ціну. Для продовження роботи необхідно натиснути кнопку "Start Project".

Після завантаження необхідних бібліотек відкриється вікно проєкту.

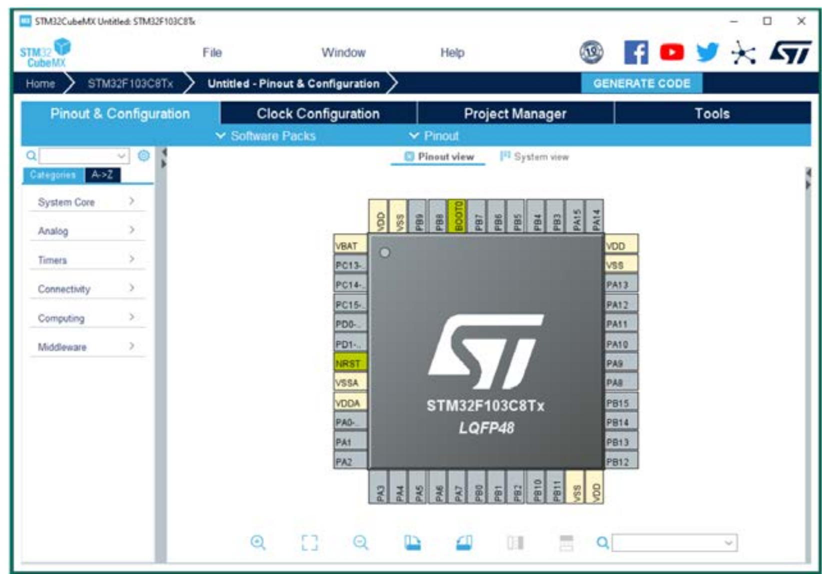


Рис. 4.27. Вигляд закладки Pinout & Configuration

Вікно проєкту містить декілька закладок та панелей. На закладці "Pinout & Configuration" розміщується панель "Categories". Основні налаштування мікроконтролера розподілені в таких категоріях: "System Core", "Analog", "Timers", "Connectivity", "Computing", "MiddleWare".

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.

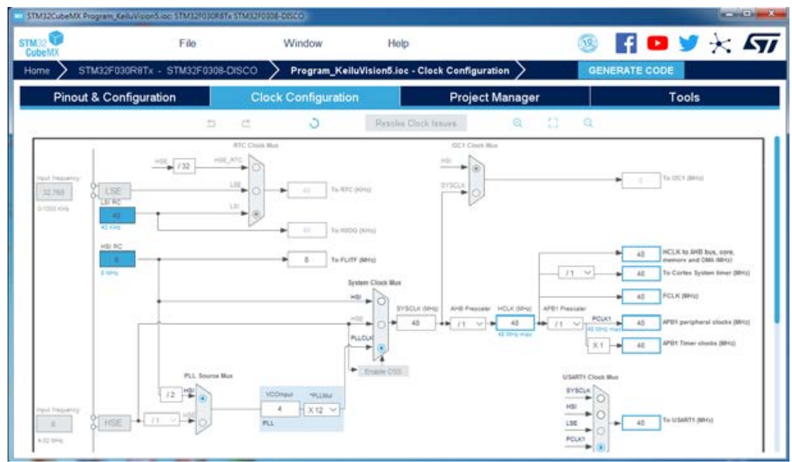


Рис. 4.28. Закладка "Clock Configuration"

Закладка "Project Manager" потрібна для вибору назви проєкту, шляху його зберігання, вибору середовища розробки (IAR™ EWARM, Keil™MDK-ARM, Atollic® TrueSTUDIO и AC6 System Workbench (SW4STM32)), під яке буде згенеровано програмний код, та параметрів налаштування програмних інструментів при генерації коду.

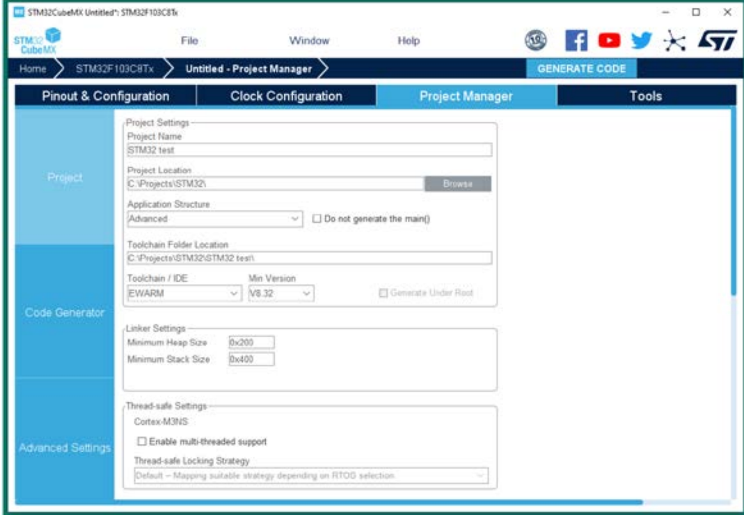


Рис. 4.29. Вигляд закладки "Project Manager"

На вкладці Code Generator слід визначитися зі складом копійованих в директорію проєкту бібліотек, налаштуваннями обробки коду користувача при регенерації (наприклад, зберігати або видаляти), налаштуваннями використання HAL. Для більш детального налаштування генератора перейдіть на вкладку Advanced Settings. Головною особливістю Сі-генератора STM32CubeMX є можливість використання як HAL-, так і LL-драйверів.

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

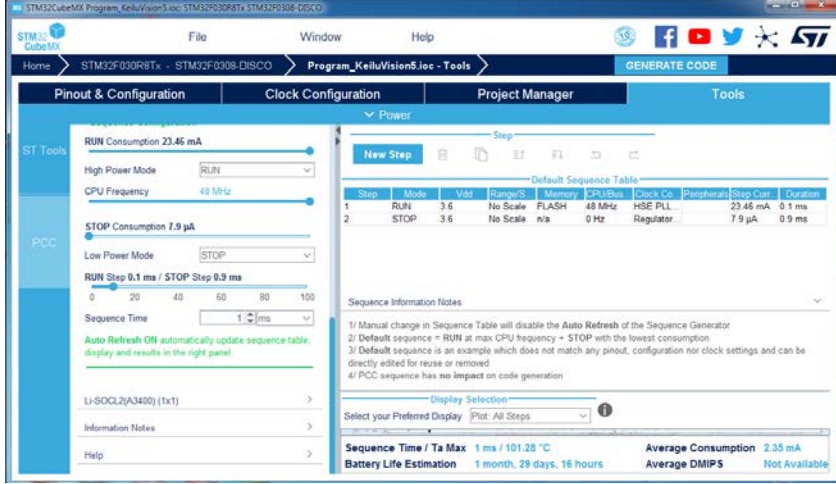


Рис. 4.30. Закладка "Tools"

Після задання потрібних налаштувань мікроконтролера, вибору назви проєкту, шляху його збереження та середовища розробки можна створити початковий код програми. Для цього потрібно натиснути кнопку "Generate Code". Внаслідок цього у зазначеній директорії проєкту буде згенеровано скелет проєкту для заданого середовища розробки.

Іноді виникає потреба міграції проєкту з однієї платформи на іншу. За допомогою STM32CubeMX це можна зробити за мінімальних витрат часу.

Примітка: В рамках одного й того самого проєкту на STM32CubeMX можна одночасно використовувати как HAL так і LL, але для різних периферійних блоків. Наприклад, налаштування Сі-генератора, при яких для UART/TIM/RTC використовуються LL-драйвери, а для решти блоків – HAL.

4.4. Програмне середовище розробки програм STM32CubeIDE

STM32CubeIDE – перше інтегроване середовище розробки від ST. Це платформа розробки C/C++ із функціями периферійної конфігурації, генерації коду, компіляції та налагодження мікроконтролерів і мікропроцесорів STM32.

Вона дає можливість розробнику вибрати мікроконтролер STM32, попередньо налаштувати проєкт, налаштувати периферію та тактування мікроконтролера, згенерувати код ініціалізації та провести компіляцію коду. STM32CubeIDE має достатній набір сучасних інструментів для налагодження. Ця платформа, створена на базі Atollic TrueStudio® для STM32 та STM32CubeMX, поєднує їх сильні сторони, тобто потужний функціонал першої та зручні та практичні утиліти другої.

Ключові особливості:

- інтегрований STM32CubeMX дозволяє:
  - здійснити вибір мікроконтролера STM32;
  - налаштувати тактування, периферію або додаткове програмне забезпечення;
  - створити проєкт та згенерувати код ініціалізації;
- STM32CubeIDE розроблена на базі ECLIPSE™/CDT та інструментальному ланцюжку GCC для розробки та GDB для налагодження. Це дозволяє інтегрувати сотні існуючих плагінів, які доповнюють функції Eclipse® IDE. Включає підтримку доповнень ECLIPSE™, компілятор GNU C/C++ і GDB-налагоджувач.
- містить аналізатори збірки та стека, які надають користувачеві корисну інформацію про статус проєкту та вимоги до пам’яті;
- включає стандартні та розширені функції налагодження:
  - перегляд регістрів периферії, ядра, пам'яті та периферійних регістрів, а також перегляд змінних у

реальному часі, інтерфейс Serial Wire Viewer або аналізатор помилок;

- системний аналіз та відстеження в реальному часі (SWV);
- інструменти, які дозволяють аналіз помилок процесора.

- Здійснюється підтримка ST-LINK та J-Link. Можливий імпорт проєкту з Atollic® TrueSTUDIO® та AC6 System Workbench STM32;
- Реалізована кросплатформність: Windows®, Linux® та macOS®.

STM32CubeIDE є універсальним інструментом, що дозволяє підвищити ефективність робочого процесу як розробників-початківців, так і для професіоналів.

4.5. Початок роботи з STM32 і STM32CubeIDE

Документація доступна як частина інсталяції STM32CubeIDE. Щоб отримати доступ до документації, перейдіть до "Довідка > Інформаційний центр". Примітки до застосування для конкретного пристрою STM32 навмисно не є частиною встановлення, оскільки вони можуть оновлюватися частіше.

4.5.1.1. Робоча область

Під час запуску STM32CubeIDE вибирається робоча область. Робоча область містить середовище розробки, яке буде використано. Технічно робоча область – це каталог, у якому можуть зберігатися проєкти. Користувач може отримати доступ до будь-якого проєкту в межах активного робочого простору.

Проєкт містить файли, які можуть бути організовані у підкаталоги. Файли, які існують десь на комп’ютері, також можна пов’язати з проєктом.

Один комп’ютер може містити кілька робочих областей у різних місцях файлової системи. Користувач може перемикатися між робочими областями, але одночасно може бути активною лише одна робоча область. Перемикання робочого простору – це швидкий спосіб переходу від одного набору проєктів до іншого.

На практиці робочий простір і модель проєкту сприяють створенню добре структурованої ієрархії робочих просторів, що містять проєкти, які, у свою чергу, містять файли.

4.5.1.2. Інформаційний центр

Під час першого запуску STM32CubeIDE і вибору робочої області відкривається Інформаційний центр. Інформаційний центр надає швидкий доступ, щоб розпочати новий проєкт, отримати доступ до відео, прочитати документацію STM32CubeIDE або отримати доступ до служби підтримки та спільноти ST. Інформаційний центр можна легко викликати в будь-який час за допомогою кнопки на панелі інструментів Інформаційного центру або з меню «Довідка».

4.5.1.3. Перспектива, меню, панель інструментів

Коли "Інформаційний центр" закрито, STM32CubeIDE відображає перспективу, яка містить рядок меню, панель інструментів, перегляди та редактори. Кожна перспектива оптимізована для особливого виду роботи. Наприклад, перспектива C/C++ призначена для створення, редагування та створення проєктів. Перспектива Debug призначена для використання під час налагодження коду на апаратному забезпеченні.

Кожну перспективу можна налаштувати відповідно до потреб користувача. Можна будь-коли скинути перспективу, якщо, наприклад, відкрито забагато переглядів або якщо перегляди змінено. Можливе й створення нової перспективи.

4.5.1.4. Вигляд і редактори.

Перспектива зазвичай відображає багато видів. Кожне представлення розроблено для надання конкретної інформації, яку, наприклад, можна зібрати з проєкту або з вбудованої системи під час налагодження. Перспектива має одну область редактора. Редактор можна використовувати для редагування файлів проєкту. Багато файлів можна редагувати в різних вкладках редактора.

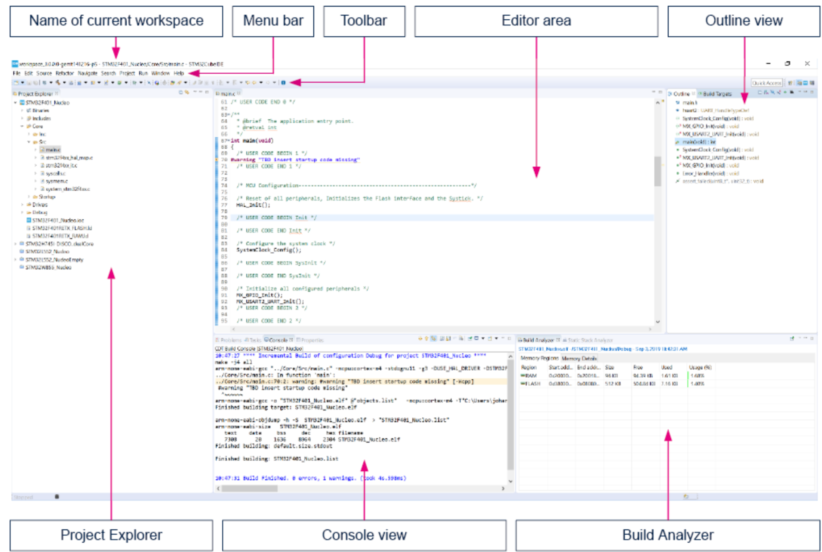


Рис. 4.31. Вікно STM32CubeIDE

4.5.2. Вступ у проєкт

Проєкт – це каталог у робочій області, що містить файли, які можуть бути організовані у підкаталоги. В активному робочому просторі можна отримати доступ до будь-якого проєкту. Файли, включені в проєкт, не обов'язково фізично розміщуються в папці проєкту, їх можна розташувати в іншому місці та зв'язати з проєктом. До проєктів, розташованих в іншому робочому просторі, неможливо отримати доступ, якщо користувач не перейде до цього робочого простору або не імпортує деякі з цих проєктів у робоче середовище, що використовується.

Є можливість перейменувати та видалити проєкт. Якщо робоча область містить багато проєктів, також можна закрити деякі з них, щоб полегшити роботу. Закриті проєкти можна знову відкрити в будь-який момент.

Цей розділ присвячено двом типам проєктів STM32, які підтримуються STM32CubeIDE:

- виконувані програми;
- статичні бібліотечні проєкти.

Однак Eclipse® C/C++ Development Toolkit (CDT™), на якому базується STM32CubeIDE, також містить базові майстри проєктів, які можна використовувати для створення проєктів керованої збірки C, керованої збірки C++ і makefile.

Проєкти STM32 можуть бути:

- C або C++;
- створений виконуваний файл або файл бібліотеки;
- на основі STM32Cube (з використанням пакета мікропрограмного забезпечення STM32) або порожніх проєктів. Проєкти STM32 також підтримують розширену парасолькову структуру проєкту, де один проєкт містить багато проєктів, наприклад, один проєкт на ядро для багатоядерних пристроїв.

4.6. Створення нового STM32 проєкту

4.6.1. Створення нового STM32 виконавчого проєкту

Найпростіший спосіб створити новий проєкт STM32 C/C++ – скористатися майстром проєкту STM32. Його вибирають через меню "Файл">"Новий проєкт STM32". Ще один спосіб створити новий проєкт C/C++ – відкрити "Інформаційний центр" і натиснути "Почати новий проєкт STM32". Інформаційний центр можна відкрити за допомогою кнопки на панелі інструментів або через меню "Довідка">"Інформаційний центр". Обидва способи ініціалізувати та запустити інструмент "STM32 Project Target Selection".

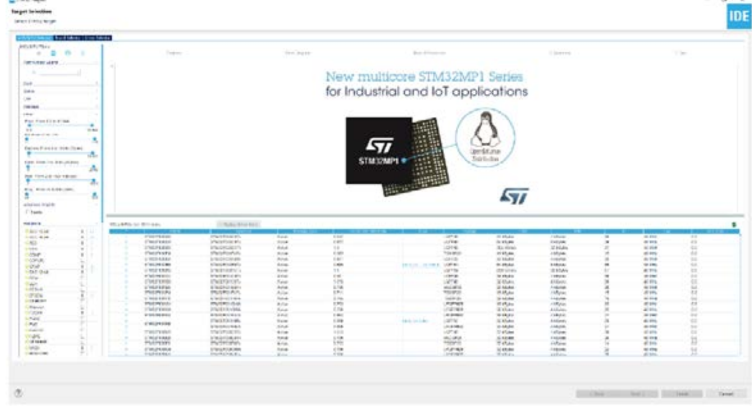


Рис. 4.32. Вибір цільового мікроконтролера STM32

Вкладки селектора "MCU/MPU" і вибору плати можна вибрати у верхній частині вікна. Використовуйте першу вкладку, щоб створити проєкт для конкретного пристрою і другий, якщо потрібен проєкт для конкретної плати. Серед різних фільтрів, доступних для використання в лівій частині вікна, введіть у полі пошуку номеру деталі самий номер деталі, щоб відфільтрувати плати з назвами, що містять цей рядок.

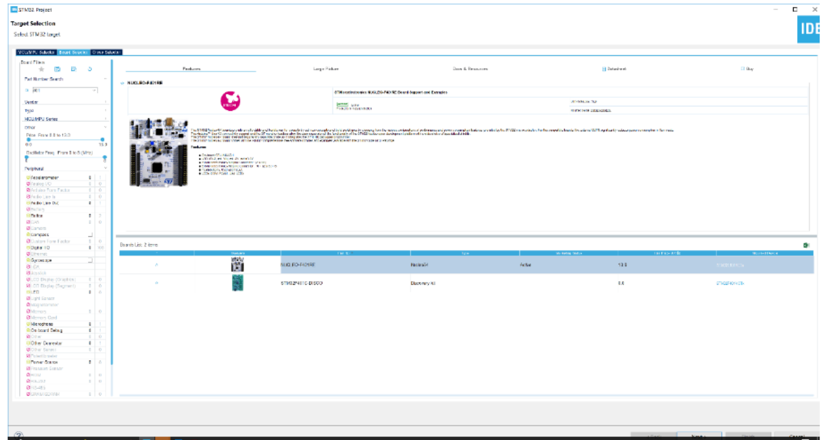


Рис. 4.33. Вибір плати STM32

П’ять вкладок – «Функції», «Великі зображення», «Документи та ресурси», «Таблиця» та «Купити» – пропонують можливість відобразити детальну інформацію про вибрану плату або пристрій. Наприклад, відображається документація, доступна для плати, і її можна відкрити, вибравши "Документи та ресурси". Якщо вибрано Datasheet, таблиця даних плати завантажується з вебсайту STMicroelectronics. Натискання "Далі", коли вибрано плату, відкриває сторінку налаштування проєкту. Введіть назву проєкту та виберіть потрібний параметр для проєкту в діалогових вікнах.

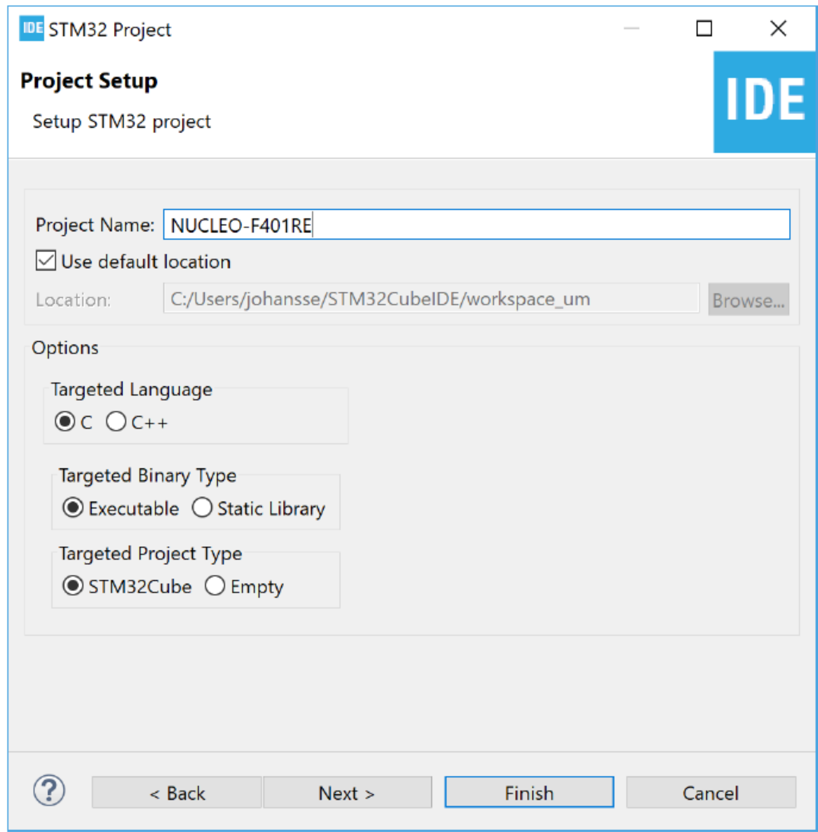


Рис. 4.34. Налаштування проєкту

Відповідно до налаштувань (рис. 4.33), проєкт має зберігатися в розташуванні за замовчуванням із такими параметрами:

- C проєкт;
- виконуваний двійковий тип;
- цільовий тип проєкту STM32Cube.

Натисніть "Далі", щоб відкрити сторінку налаштування пакета бібліотеки прошивки.

Рис. 4.35. Налаштування пакета бібліотеки прошивки

На цій сторінці можна вибрати пакет мікропрограми STM32Cube для використання під час створення проєкту. У цьому випадку використовуються параметри за замовчуванням. Натисніть "Готово", щоб створити проєкт. У результаті відобразиться таке діалогове вікно.

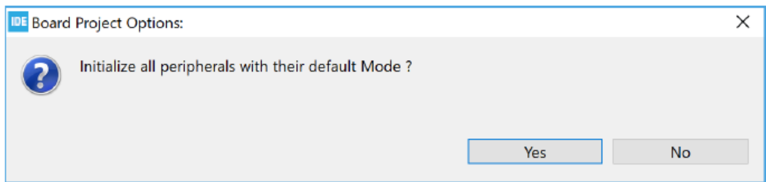


Рис. 4.36. Ініціалізація периферійних пристроїв

Натисніть "Так", щоб отримати програмне забезпечення, необхідне для ініціалізації периферійних пристроїв. Відкриється нове діалогове вікно, показане на рис. 4.36.

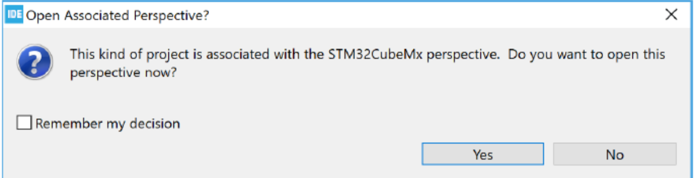


Рис. 4.37. Відкриття перспективи STM32CubeMX

Відкриття перспективи STM32CubeMX – добре рішення, якщо є потреба налаштувати пристрій. Увімкніть "Запам'ятати моє рішення", якщо запитання не буде задано під час наступного створення нового проєкту. Натисніть "Так", щоб продовжити. На цьому етапі починається створення проєкту. Час створення залежить від кількості файлів, які необхідно завантажити для проєкту.

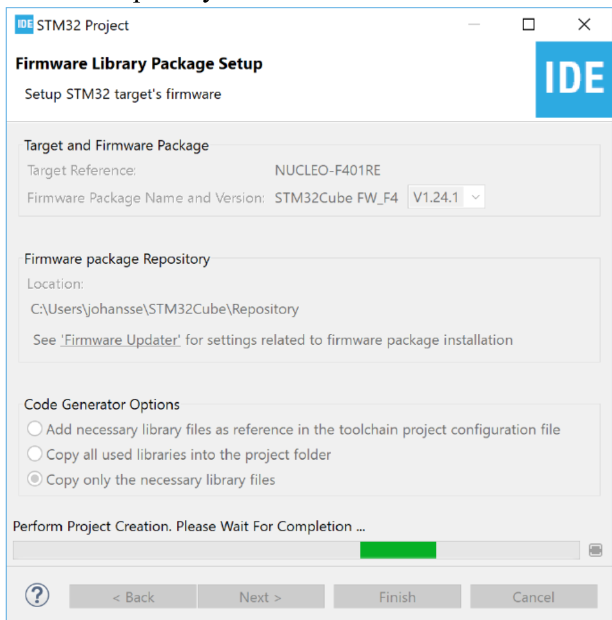


Рис. 4.38. Вікно створення нового проєкту

Коли проєкт створено, відкривається перспектива STM32CubeMX із вікном для налаштування периферійних пристроїв, годинника, проміжного програмного забезпечення та енергоспоживання.

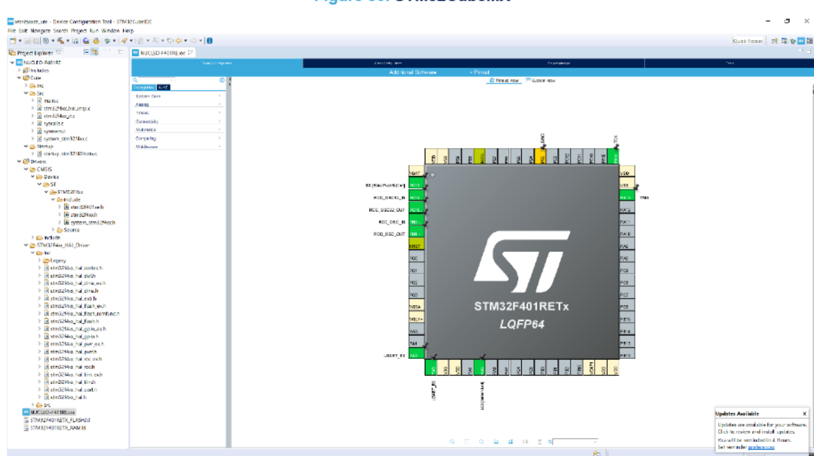


Рис. 4.39. Загальний вигляд STM32CubeMX

Новий проєкт відображається у вікні Project Explorer разом із деякими папками та файлами, які він містить.

Файл *.ioc містить налаштування конфігурації, відкривається в редакторі STM32CubeMX.

Цей редактор містить вкладки для розпіновки та конфігурації, конфігурації годинника, менеджера проєкту та інструментів. Коли зміни вносяться в редактор STM32CubeMX, файл *.ioc на вкладці позначається як змінений. Якщо файл збережено, відкриється діалогове вікно із запитанням «Ви хочете створити код?», що полегшить створення нового коду в проєкті, який підтримує нову конфігурацію пристрою. Для отримання додаткової інформації про використання редактора STM32CubeMX потрібно звернутися до документації.

Можна створити проєкт STM32 із меншою кількістю файлів і папок, вибравши цільовий тип проєкту "Empty" замість "STM32Cube" (див. рис. 4.33 Налаштування проєкту). Якщо вибрано "Empty", згенерований проєкт містить лише деякі

папки, файл запуску пристрою з кодом Reset_Handler і векторною таблицею, файл main.c та деякі інші файли *.c і файли сценарію компонування. Файли заголовків STM32, системні файли та файли CMSIS потрібно додавати вручну. Ці файли можна, наприклад, скопіювати з іншого цільового проєкту STM32Cube або з прикладу проєкту STM32.

Примітка: для порожніх проєктів обов’язково впевніться в налаштуваннях блока з плаваючою комою на використання програмного FPU або апаратного FPU відповідно до вимог програми. У разі використання апаратного FPU ініціалізуйте FPU.

Для непорожніх проєктів ініціалізація FPU зазвичай виконується у функції System Init у файлі system_stm32fxxx.c. Щоб сповістити про те, що може знадобитися конфігурація FPU, файл main.c, створений у порожньому проєкті, містить попередження компілятора #warning "FPU не ініціалізовано, але проєкт компілюється для FPU. Ініціалізуйте FPU перед використанням".

4.6.2. Створення нового STM32 проєкту статичної бібліотеки

Метод, описаний у пункті 4.6.1 про створення нового виконуваного проєкту STM32, також можна використовувати для створення проєкту статичної бібліотеки. Однак статичні бібліотеки часто повторно використовуються в проєктах кількох програм, можливо, націлених на різні продукти STM32. Майстер проєкту STM32 не підтримує зміну пристрою MCU. Стандартне рішення проєктів Eclipse®/CDT™ із деякими розширеннями, пов’язаними з STM32, підтримує зміну пристрою MCU для кожної конфігурації збірки.

Рекомендований спосіб створення проєктів статичної бібліотеки полягає в тому, щоб покладатися на проєкти Eclipse®/CDT™, як описано нижче.

Щоб створити Eclipse®/CDT™, перейдіть до "Файл">"Новий">"C/C++ Project". Відкриється вікно, зображене на рис. 4.39.

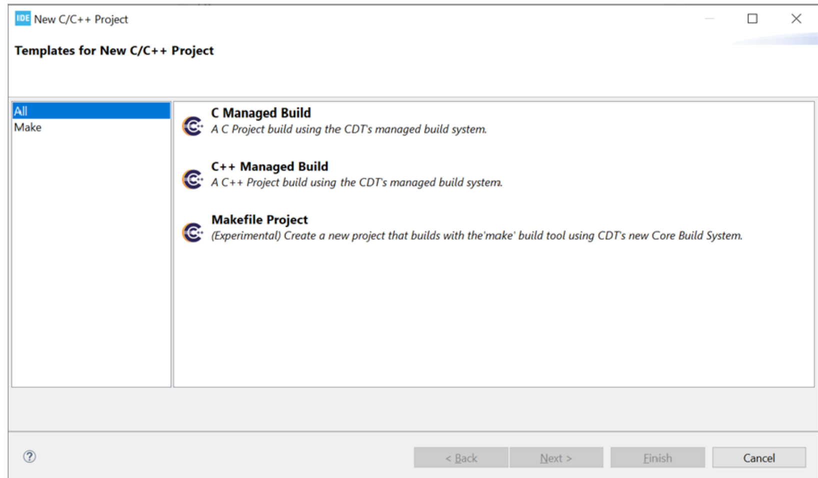


Рис. 4.40. Новий C/C++ проєкт

Виберіть C Managed Build або C++ Managed Build залежно від того, що вимагає проєкт, і натисніть "Далі".

Це відкриває селектор типу проєкту. Тип "Empty Project" – це єдиний тип, який підтримує ланцюжок інструментів MCU ARM GCC. Обов'язково виберіть "Порожній проєкт" у папці "Виконуваний файл", а потім виберіть ланцюжок інструментів MCU ARM GCC, як показано на рис. 4.40.

Після того, як назва проєкту та вибір типу завершені, натисніть "Далі".

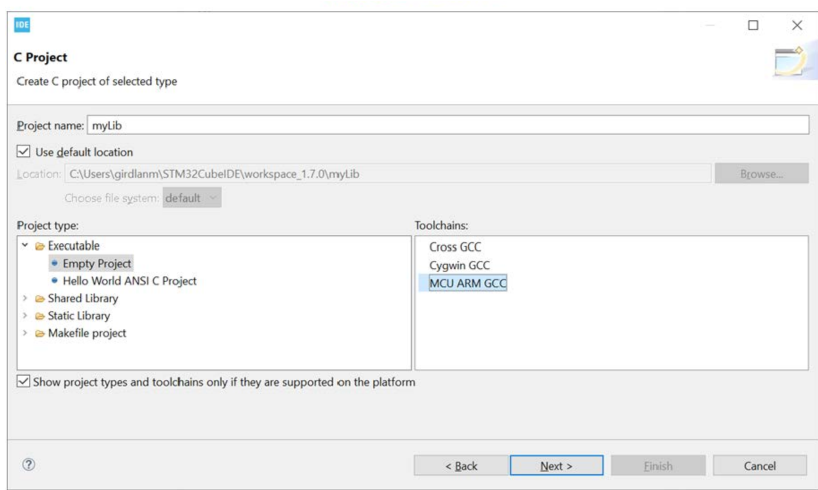


Рис. 4.41. Тип проєкту

Це запустить стандартне вікно конфігурації проєкту Eclipse®, як показано на рис. 4.41. Натисніть "Далі".

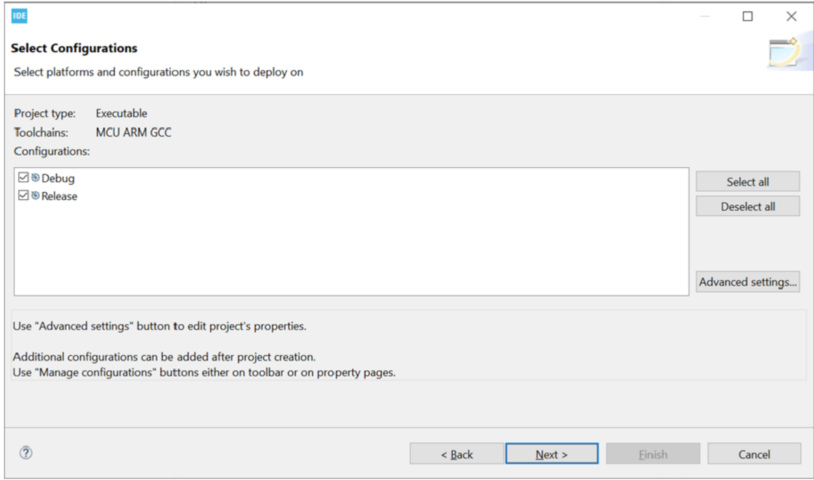


Рис. 4.42. Вікно конфігурації проєкту

На екрані вибору цілей (рис. 4.42) переконайтеся, що ви вибрали відповідний цільовий мікроконтролер, натиснувши кнопку "Вибрати…" і відфільтрувавши правильну ціль для

проєкту. Селектор цілей допомагає правильно встановити прапорець ланцюжка інструментів -mcpu=cortex-mX у вже визначених конфігураціях збірки, які ми бачили на попередньому кроці (див. рис. 4.41).

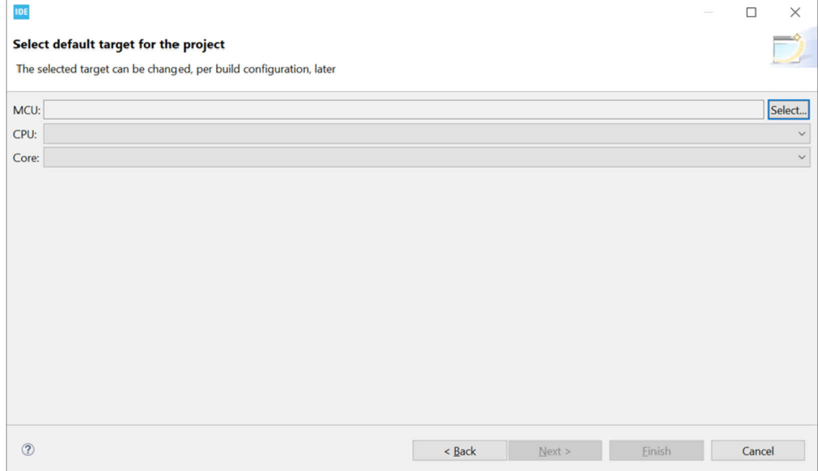


Рис. 4.43. Вікно вибору цільових пристроїв

Якщо натиснути "Вибрати...", з’явиться діалогове вікно фільтра (рис. 4.43), що дозволяє користувачам фільтрувати та вибирати правильний пристрій.

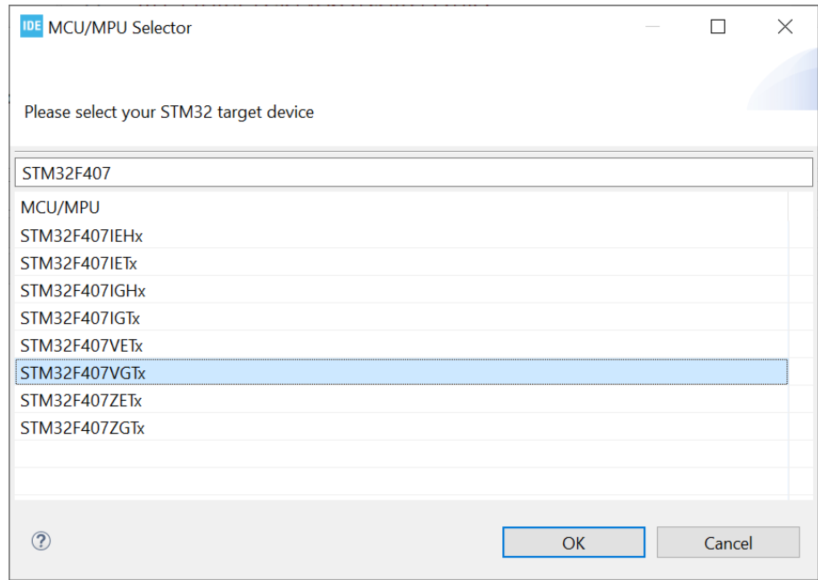


Рис. 4.44. Вікно вибору MCU/MPU

Після вибору цілі (MCU) поля CPU і Core автоматично заповнюються в простому одноядерному корпусі (рис. 4.44).

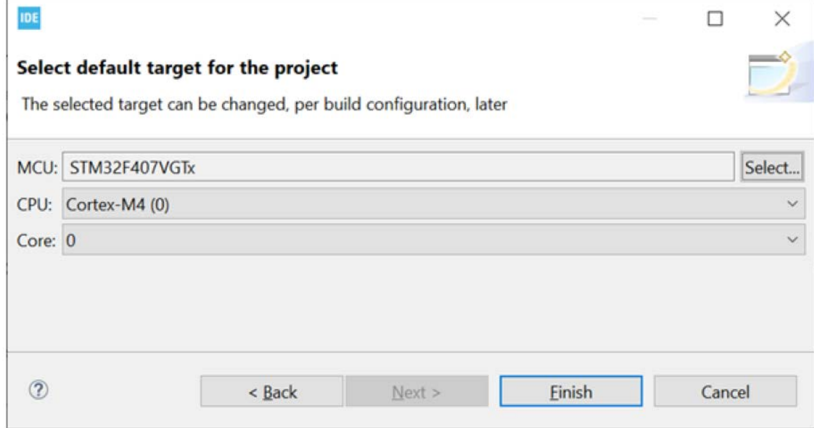


Рис. 4.45. Вибір цільового пристрою

Для більш сучасних пристроїв, таких як багатоядерні мікроконтролери STM32H7, користувач повинен вибрати ЦП і Ядро, на які націлений проєкт, щоб переконатися, що код побудовано правильно. Ці параметри також використовуються пізніше для правильного налаштування конфігурацій налагодження. Переконайтеся, що параметри потрібні для проєкту, і натисніть "Готово".

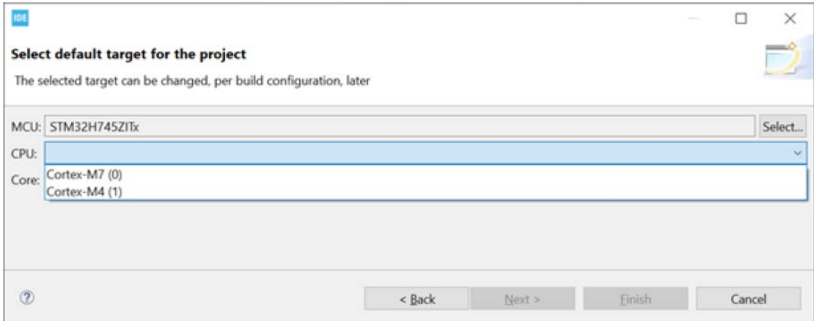


Рис. 4.46. Вибір ядра цільового пристрою

Після створення проєкту можна створювати різні конфігурації збірки для різних цільових контролерів. Після створення нової конфігурації збірки клацніть правою кнопкою миші проєкт у Провіднику проєктів, перейдіть до "Властивості" > "Складання C/C++" > "Налаштування" > "Параметри інструмента" > "Параметри MCU" і натисніть "Вибрати ...", щоб вибрати нову ціль для певної конфігурації збірки.

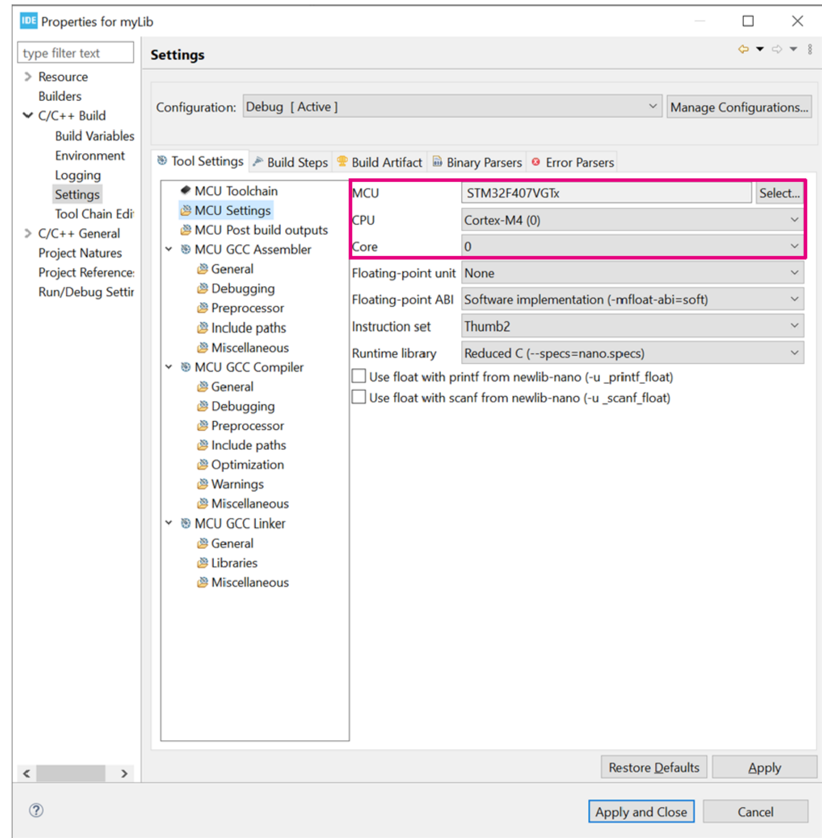


Рис. 4.47. Зміна цільового пристрою

Порада: проєкти CDT™ також можна використовувати для створення цільових незалежних програм або виконуваних проєктів.

4.7. Приклад створення проєкту в середовищі STM32CubeIDE

При першому запуску програма запропонує позначити папку, де зберігаються проєкти. Далі тиснемо "Start new STM32 project" або через ("File"->"New"->"STM32 Project").

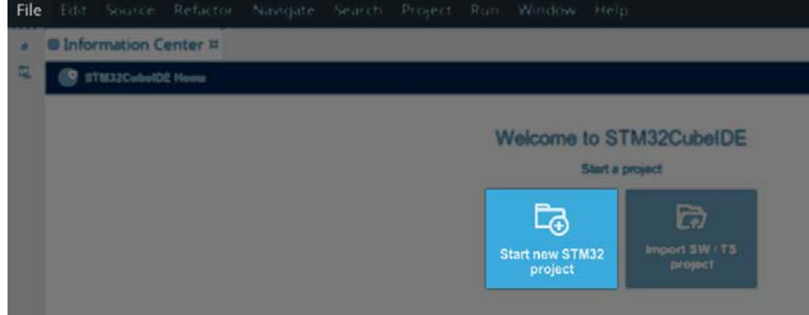


Рис. 4.48. Початок нового проєкту

Вибираємо необхідний мікроконтролер і тиснемо "Next".

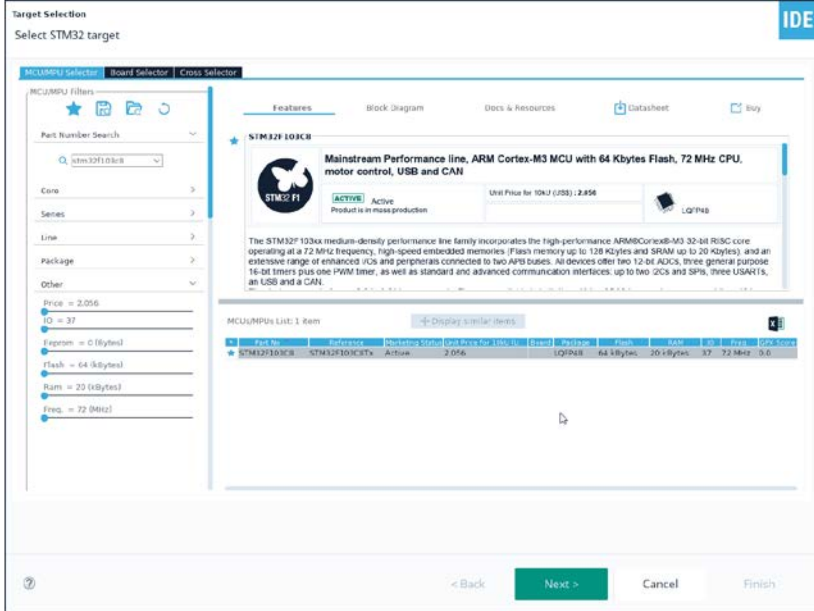


Рис. 4.49. Вибір необхідного мікроконтролера

Даємо ім'я проєкту і тиснемо "Finish".

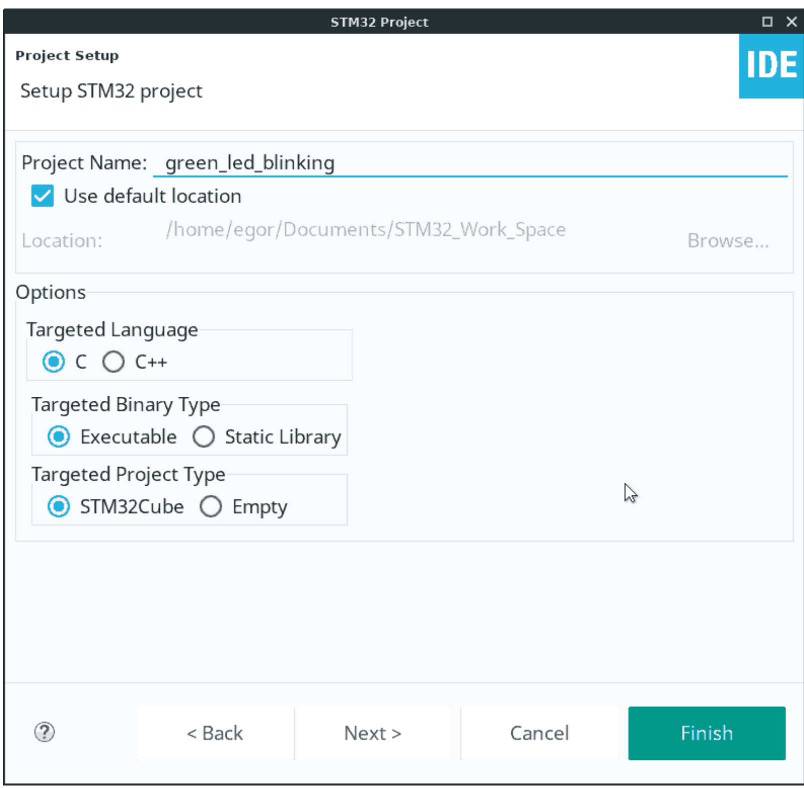


Рис. 4.50. Додавання імені проєкту

Вмикаємо спосіб налагодження (наприклад, послідовне налагодження по SWD "Serial Wire Debug", щоби використовувати ST-LINK, меню "SYS" -> "Debug" -> "Serial Wire").

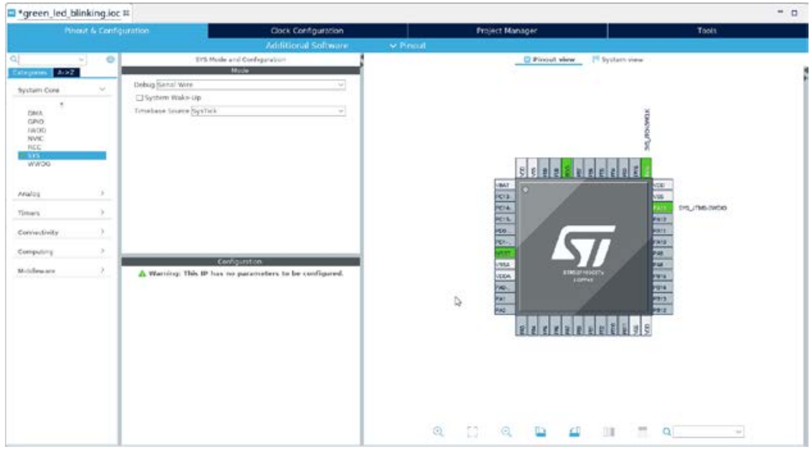


Рис. 4.51 Вибір способу налагодження

Вибираємо джерело тактування "RCC" -> "HSE (High Speed Clock)" -> "Crystal/Ceramic Resonator".

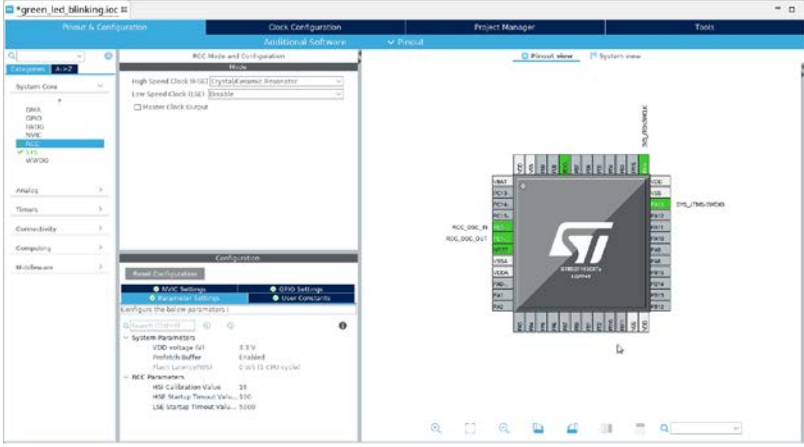


Рис. 4.52 Вибір джерела тактування

У вкладці налаштування тактової системи вводимо частоту МГц і підтверджуємо запропоноване рішення.

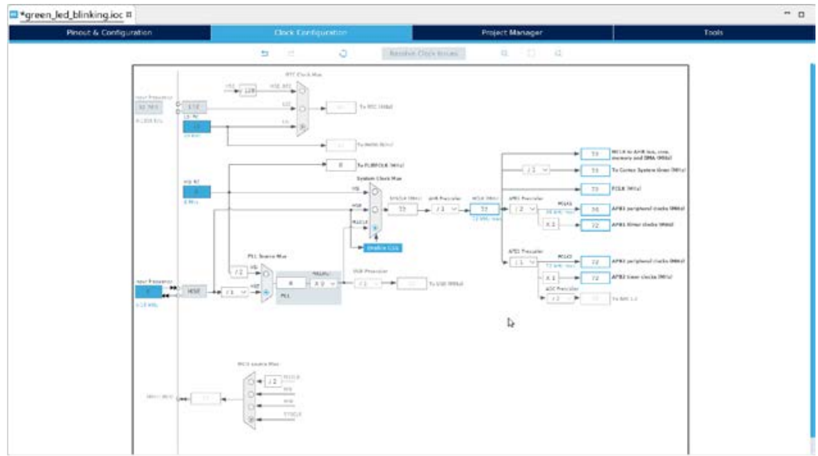


Рис. 4.53 Задання тактової частоти

Налаштовуємо виводи.

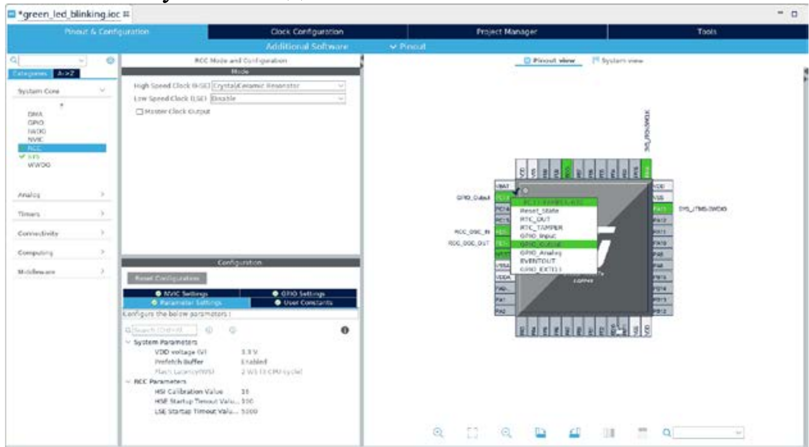


Рис. 4.54 Налаштування виводів

Далі ліворуч у дереві проєкту в папці "Src" робимо подвійне клацання по головному файлу main.c

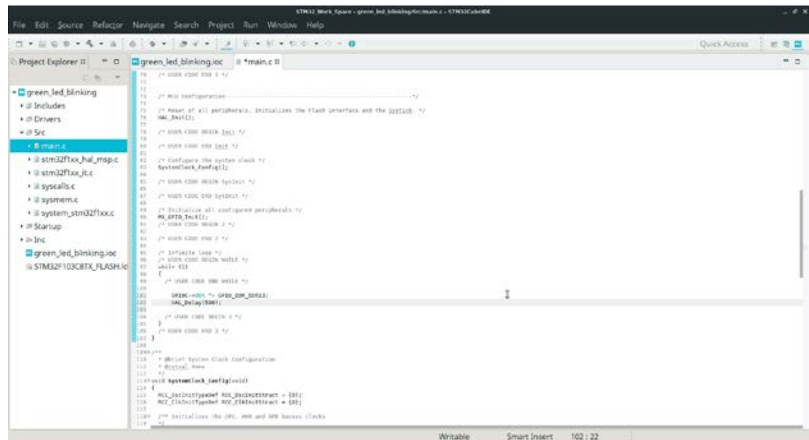


Рис. 4.55 Вікно з підготовленим кодом програми

Пишемо код програми. Збираємо проєкт, натиснувши комбінацію клавіш "Ctrl+B" або клацнувши по іконці "молоточок". Якщо є помилки або попередження в проєкті – виправляємо і знову проводимо збирання.

Рис. 4.56 Вікно виведення повідомлень компіляції

Запускаємо налагодження (F11 або "Run" -> "Debug"). Перемикаємось у простір налагодження.

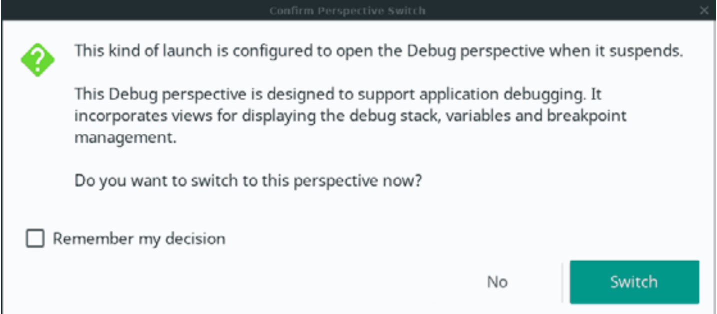


Рис. 4.57 Перемикання в простір налагодження

Можливо, доведеться обновити прошивку програматора-налагоджувача.

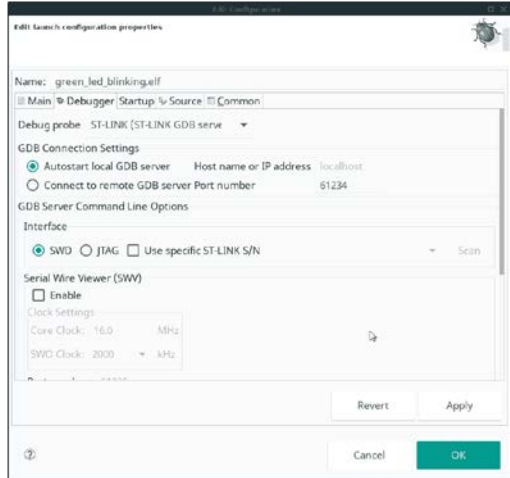


Рис. 4.58 Обновлення прошивки програматора

Далі здійснюємо завантаження прошивки в мікроконтролер.

Рис. 4.59 "Прошивання" програми в мікроконтролер

Натискаючи на кнопки продовження/паузи керуємо виконанням програми.

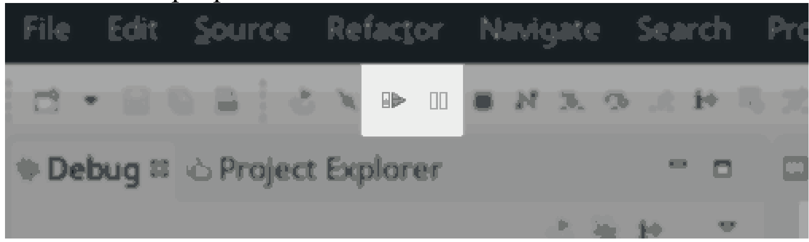


Рис. 4.60 Керування роботою програми

Можемо спостерігати за значеннями регистрах.

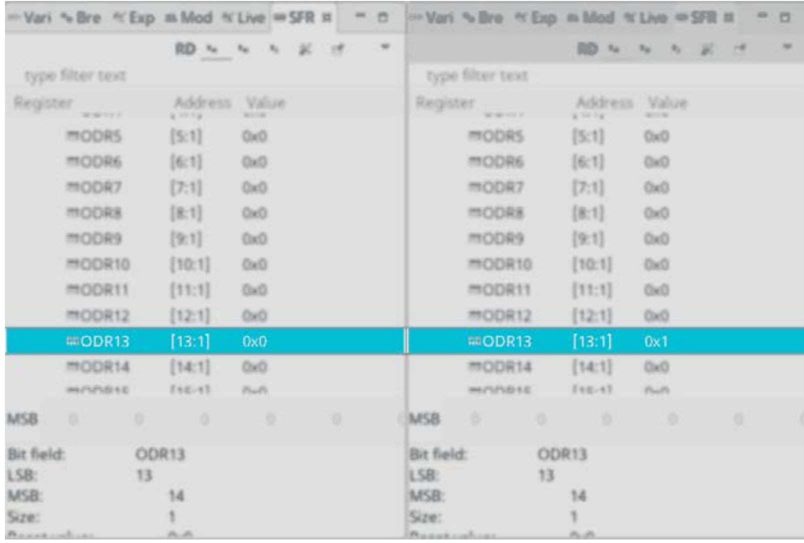


Рис. 4.61 Вміст регістрів

4.8. *Література до розділу*

1. https://ela.kpi.ua/bitstream/123456789/48924/1/Osnovy_proektuvannia_system_Internetu_rechei.pdf
2. http://indrekis2.blogspot.com/2012/10/stm32-arm-cortex-m-stmicroelectronics.html
3. https://ritorika.com.ua/instrukcija/30/sreda-razrabotki-dlja-stm32-kakuju-vybrat
4. колекція посилань про роботу із IDE:
5. CoIDE: друга частина "STM32: Урок 1 - Настраиваем IDE" присв'ячена, власне, роботі з цим IDE; "Необходимый софт" від EasySTM32.
6. Keil: "ARM. Учебный Курс. Keil + CMSIS. Создание проєкта" (від Ді Халта); "Встроенный ST-LINK + STM32F103 + Keil" (від ChipSpace).
7. IAR EWARM: "ARM. Учебный курс. IAR EWARM. Создание проєкта часть 2. CMSIS и Standard Peripherals Library" (від Ді Халта); "STM32. Создание проєкта в IAR EWARM. Работа с портами ввода/вывода. Часть 1.", "Часть 2", "Подключение CMSIS в IAR EWARM с версии 6.2" (від ChipSpace).
8. www.iar.com

9. www.keil.com
10. www.raisonance.com
11. www.atollic.com
12. www.eclipse.org
13. https://learntutorials.net/ru/stm32/topic/7741/интегрированные-среды-разработки--ide-
14. https://www.compel.ru/lib/53954
15. "Operating Systems and Middleware" від STMicroelectronics
16. "Open source Real time Operating Systems for the STM32 and Cortex m3 MCu's"
17. "List of real-time operating systems" на Wiki
18. https://blog.avislab.com/keil/
19. https://www.st.com/resource/en/user_manual/dm00598966-stm32cubeide-quick-start-guide-stmicroelectronics.pdf
20. https://www.youtube.com/watch?v=eumKLXNlM0U
21. https://ritorika.com.ua/instrukcija/30/sreda-razrabotki-dlja-stm32-kakuju-vybrat
22. https://www.st.com/en/development-tools/stm32cubeide.html
23. https://www.compel.ru/lib/125813
24. https://wiki.stmicroelectronics.cn/stm32mcu/wiki/STM32CubeIDE:Introduction_to_STM32CubeIDE
25. https://www.st.com/resource/en/user_manual/dm00629856-description-of-the-integrated-development-environment-for-stm32-products-stmicroelectronics.pdf
26. https://cxemka.com/24-sozdanie-proekta-v-srede-stm32cubeide.html
27. https://www.ecb.torontomu.ca/~courses/coe718/F22-labs/lab1.pdf

РОЗДІЛ 5. Приклади програм для STM32

5.1. Використання портів введення/виведення

Розглянемо приклад програми, яка демонструє роботу з портами вводу/виводу. Вона дозволяє перемикати стани світлодіода при натисканні кнопки користувача.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.h"
#include "stm32f4xx_gpio.h"

void Delay(uint32_t nCount)
{
while(nCount--) { }
}

int main(void)
{
GPIO_InitTypeDef gpioConf;
// инициализация входа, подключенного к кнопке
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
gpioConf.GPIO_Pin = GPIO_Pin_0;
gpioConf.GPIO_Mode = GPIO_Mode_IN;
GPIO_Init(GPIOA, &gpioConf);
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
// инициализация входа, подключенного к светодиоду
gpioConf.GPIO_Pin = GPIO_Pin_13;
gpioConf.GPIO_Mode = GPIO_Mode_OUT;
gpioConf.GPIO_Speed = GPIO_Speed_100MHz;
gpioConf.GPIO_OType = GPIO_OType_PP;
gpioConf.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOD, &gpioConf);

while(1)
    {
    if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0) == 0)
        {
        if (GPIO_ReadOutputDataBit(GPIOD, GPIO_Pin_13))
            GPIO_ResetBits(GPIOD, GPIO_Pin_13);
        else
            GPIO_SetBits(GPIOD, GPIO_Pin_13);
        Delay(5000);
        while(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)==0);
        Delay(5000);
```

```
        }
    }
}
```

5.2. Переривання та їх використання. Використання таймерів

Ця програма демонструє роботу з перериваннями та таймерами, описані функції, перерахування, структури для роботи з перериваннями. У ній реалізовано перемикання світлодіода, підключеного до порту вводу/виводу, через певні інтервали часу.

```
#include «stm32f4xx.h»
#include "stm32f4xx_gpio.h"
#include "stm32f4xx_rcc.h"
#include "stm32f4xx_tim.h"
#include "misc.h"
void INTTIM_Config(void);
void GPIO_Config(void);

int main(void)
{
GPIO_Config();
INTTIM_Config();
while(1) { }
}

void TIM2_IRQHandler(void)
{
if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
    {
    TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
    GPIOD->ODR ^= GPIO_Pin_13;
    }
}

void GPIO_Config(void)
{
GPIO_InitTypeDef gpio_struct;
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
gpio_struct.GPIO_Pin = GPIO_Pin_13;
gpio_struct.GPIO_Mode = GPIO_Mode_OUT;
gpio_struct.GPIO_OType = GPIO_OType_PP;
gpio_struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
gpio_struct.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_Init(GPIOD, &gpio_struct);
```

```
}

void INTTIM_Config(void)
{
NVIC_InitTypeDef nvic_struct;
nvic_struct.NVIC_IRQChannel = TIM2_IRQn;
nvic_struct.NVIC_IRQChannelPreemptionPriority = 0;
nvic_struct.NVIC_IRQChannelSubPriority = 1;
nvic_struct.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&nvic_struct);

TIM_TimeBaseInitTypeDef tim_struct;
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
tim_struct.TIM_Period = 10000 - 1;
tim_struct.TIM_Prescaler = 168 - 1;
tim_struct.TIM_ClockDivision = 0;
tim_struct.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM2, &tim_struct);
TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
TIM_Cmd(TIM2, ENABLE);
}
```

5.3. Генерація сигналу ШІМ

Для ознайомлення з можливістю генерації сигналу ШІМ та використання його для керування яскравістю світлодіодів наведена наступна програма. У лістингу представлено зміст файлу із головною функцією.

```
#include "stm32f4xx.h"
#include "stm32f4xx_gpio.h"
#include "stm32f4xx_rcc.h"
#include "misc.h"
#include "stm32f4xx_tim.h"
#include "init.h"

int main(void)
{
init();

int brightness = 0;
int i;
while(1)
    {
    brightness++;

    TIM4->CCR3 = 333 - (brightness + 0) % 333;
    TIM4->CCR4 = 333 - (brightness + 166 / 2) % 333;
```

```
    TIM4->CCR1 = 333 - (brightness + 333 / 2) % 333;
    TIM4->CCR2 = 333 - (brightness + 499 / 2) % 333;

    for(i=0;i < 10000; ++i);
    for(i=0;i < 10000; ++i);
    for(i=0;i < 10000; ++i);
    }
}
```

В основній функції відбувається зміна значень регістра захоплення/порівняння з певною затримкою.

Функції з виконанням налаштувань перенесені до окремого файлу init.c. Відповідно оголошення функцій знаходяться у файлі init.h. Наступний список демонструє вміст файлу init.c.

```
#include "init.h"
#include "stm32f4xx.h"
#include "stm32f4xx_gpio.h"
#include "stm32f4xx_rcc.h"
#include "misc.h"
#include "stm32f4xx_tim.h"
void init() {
GPIOinit();
TimerInit();
}

void TimerInit() {
TIM_TimeBaseInitTypeDef time_init;
TIM_OCInitTypeDef oc_init;
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);

uint16_t PrescalerValue = (uint16_t)((SystemCoreClock / 2) / 21000000);

time_init.TIM_Period = 665;
time_init.TIM_Prescaler = PrescalerValue;
time_init.TIM_ClockDivision = 0;
time_init.TIM_CounterMode = TIM_CounterMode_Up;

TIM_TimeBaseInit(TIM4, &time_init);

oc_init.TIM_OCMode = TIM_OCMode_PWM1;
oc_init.TIM_OutputState = TIM_OutputState_Enable;
oc_init.TIM_Pulse = 0;
oc_init.TIM_OCPolarity = TIM_OCPolarity_High;

TIM_OC1Init(TIM4, &oc_init);
TIM_OC1PreloadConfig(TIM4, TIM_OCPreload_Enable);
```

```
oc_init.TIM_OutputState = TIM_OutputState_Enable;
oc_init.TIM_Pulse = 0;

TIM_OC2Init(TIM4, &oc_init);
TIM_OC2PreloadConfig(TIM4, TIM_OCPreload_Enable);

oc_init.TIM_OutputState = TIM_OutputState_Enable;
oc_init.TIM_Pulse = 0;

TIM_OC3Init(TIM4, &oc_init);
TIM_OC3PreloadConfig(TIM4, TIM_OCPreload_Enable);

oc_init.TIM_OutputState = TIM_OutputState_Enable;
oc_init.TIM_Pulse = 0;

TIM_OC4Init(TIM4, &oc_init);
TIM_OC4PreloadConfig(TIM4, TIM_OCPreload_Enable);

TIM_ARRPreloadConfig(TIM4, ENABLE);
TIM_Cmd(TIM4, ENABLE);
}

void GPIOinit() {
GPIO_InitTypeDef gpio_init;
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);

gpio_init.GPIO_Pin=GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_1
5;
gpio_init.GPIO_Mode = GPIO_Mode_AF;
gpio_init.GPIO_Speed = GPIO_Speed_100MHz;
gpio_init.GPIO_OType = GPIO_OType_PP;
gpio_init.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_Init(GPIOD, &gpio_init);

GPIO_PinAFConfig(GPIOD, GPIO_PinSource12, GPIO_AF_TIM4);
PIO_PinAFConfig(GPIOD, GPIO_PinSource13, GPIO_AF_TIM4);
GPIO_PinAFConfig(GPIOD, GPIO_PinSource14, GPIO_AF_TIM4);
GPIO_PinAFConfig(GPIOD, GPIO_PinSource15, GPIO_AF_TIM4);
}
```

Як бачимо з прикладу, для ініціалізації таймера необхідно відразу дві структури: одна – для ініціалізації безпосередньо таймера, а інша – для задання режиму порівняння виходу. Наведена програма дозволяє послідовно зменшувати яскравість світіння світлодіодів користувача на платі.

5.4. Використання АЦП

Вміст основного файлу main.c наведено нижче.

```
#include <stm32f4xx_rcc.h>
#include <stm32f4xx_gpio.h>
#include <stm32f4xx_adc.h>

void leds_init() {
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

GPIO_StructInit(&gpio);
gpio.GPIO_Mode = GPIO_Mode_AN;
gpio.GPIO_Pin = GPIO_Pin_1;
GPIO_Init(GPIOA, &gpio);
}

void adc_init() {
ADC_InitTypeDef ADC_InitStructure;
ADC_CommonInitTypeDef adc_init;
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
ADC_DeInit();

ADC_StructInit(&ADC_InitStructure);
adc_init.ADC_Mode = ADC_Mode_Independent;
adc_init.ADC_Prescaler = ADC_Prescaler_Div2;

ADC_InitStructure.ADC_ScanConvMode = DISABLE;
ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
ADC_InitStructure.ADC_ExternalTrigConv =
ADC_ExternalTrigConvEdge_None;
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;

ADC_CommonInit(&adc_init);
ADC_Init(ADC1, &ADC_InitStructure);
ADC_Cmd(ADC1, ENABLE);
}

u16 readADC1(u8 channel) {
ADC_RegularChannelConfig(ADC1, channel, 1, DC_SampleTime_3Cycles);
ADC_SoftwareStartConv(ADC1);
while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);
return ADC_GetConversionValue(ADC1);
}

void Delay(unsigned int Val) {
for (; Val != 0; Val--);
```

```
}

int main(void) {
leds_init();
adc_init();

do {
    unsigned int bin_code = readADC1(ADC_Channel_1);
    double voltage = bin_code * 2.96 / 0xfff;

    Delay(500000);
    } while (1);
}
```

У ній є кілька моментів, на які слід звернути особливу увагу. По-перше, при ініціалізації порту ми задаємо значення, яке потрібно для роботи в аналоговому режимі за допомогою значення GPIO_Mode_AN. По-друге, ініціалізація та зчитування значень із АЦП. Ми налаштовуємо АЦП, що перетворення здійснюється програмно, і виконуємо його за допомогою виклику функції readADC1.

5.5. Використання USART

Розглянемо приймання даних за допомогою USART1. Для передачі та приймання використовуються виходи порту А під номерами 9 (Transmit) та 10 (Receive). Для кожного приймача-передавача виділено свої пари виводів для приймання та передавання, які можна знайти у документації до мікроконтролера. Насамперед, включаємо тактування необхідних пристроїв.

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
GPIO_InitStruct.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_Init(GPIOA, &GPIO_InitStruct);
```

Далі вибираємо альтернативну функцію, що виконується виводами порту. Робимо це за допомогою функції GPIO_PinAFConfig().

```
GPIO_PinAFConfig(GPIOA, GPIO_PinSource9, GPIO_AF_USART1);
GPIO_PinAFConfig(GPIOA, GPIO_PinSource10, GPIO_AF_USART1);
```

Слід звернути увагу, що налаштуванню підлягають лише певні виводи, інші виводи, наприклад PA2 і PA3, використовувати USART як виводів не вдасться.

Далі необхідно ініціалізувати USART. Код ініціалізації дуже схожий на ініціалізацію порту.

```
USART_InitStruct.USART_BaudRate = baudrate;
USART_InitStruct.USART_WordLength = USART_WordLength_8b;
USART_InitStruct.USART_StopBits = USART_StopBits_1;
USART_InitStruct.USART_Parity = USART_Parity_No;
USART_InitStruct.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;
USART_InitStruct.USART_Mode = USART_Mode_Rx;
USART_Init(USART1, &USART_InitStruct);
```

На останньому етапі ініціалізації необхідно налаштувати переривання та увімкнути USART.

```
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
USART_Cmd(USART1, ENABLE);
```

Після того, як ми увімкнули переривання, необхідно визначити обробник цього переривання. Обробник переривання має USART1_IRQHandler і він не повертає і не приймає жодних даних. Крім того, USART підтримує кілька переривань і для всіх переривань використовується один обробник. Тому необхідно перевіряти прапор (RXNE) у регістрі статусу, щоб організувати обробку різних переривань у різних блоках коду обробника. Після перевірки стану прапора можна скинути прапор RXNE. Далі зчитуємо дані з регістру даних та виводимо їх на дисплей.

```
void USART1_IRQHandler(void){
if( USART_GetITStatus(USART1, USART_IT_RXNE) ) {
    USART_ClearITPendingBit(USART1, USART_IT_RXNE);
    static uint8_t cnt = 0;
    uint8_t t = USART_ReceiveData(USART1);
    write_char(t);
    }
}
```

5.6. *Робота з SPI*

Інтерфейс SPI для організації передавання даних між пристроями, дослідження режиму ведучого та веденого.

```
#include <stm32f4xx.h>
#include <stm32f4xx_rcc.h>
#include <stm32f4xx_gpio.h>
#include <stm32f4xx_spi.h>
#include <misc.h>

#define SPI_PINS GPIO_Pin_5 | \
     GPIO_Pin_6 | \
     GPIO_Pin_7

void init(void);
void delay(int count);

void SPI1_IRQHandler(void) {
int res;
if (SPI_I2S_GetITStatus(SPI1, SPI_I2S_IT_RXNE) != RESET) {
     SPI_I2S_ClearITPendingBit(SPI1, SPI_I2S_IT_RXNE);
     res = SPI_I2S_ReceiveData(SPI1);
     }
}

int main(void)
{
init();
int sendData = 0;
while(1)
     {
     delay(100);
     SPI_I2S_SendData(SPI2, sendData);
     while (SPI_I2S_GetFlagStatus(SPI2, SPI_I2S_FLAG_TXE) == RESET);
     sendData++;
     if (sendData == 0xff) sendData = 0;
     }
}

void init(void) {
GPIO_InitTypeDef gpio_init;
SPI_InitTypeDef spi_init;
NVIC_InitTypeDef nvic_init;

RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA |
RCC_AHB1Periph_GPIOB |
RCC_AHB1Periph_GPIOC, ENABLE);

 /* Configure slave pins */
gpio_init.GPIO_Mode = GPIO_Mode_AF;
```

```
gpio_init.GPIO_Pin = SPI_PINS;
gpio_init.GPIO_Speed = GPIO_Speed_50MHz;
gpio_init.GPIO_OType = GPIO_OType_PP;
gpio_init.GPIO_PuPd = GPIO_PuPd_DOWN;
GPIO_Init(GPIOA, &gpio_init);

GPIO_PinAFConfig(GPIOA, GPIO_PinSource5, GPIO_AF_SPI1);
GPIO_PinAFConfig(GPIOA, GPIO_PinSource6, GPIO_AF_SPI1);
GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_SPI1);

gpio_init.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3;
GPIO_Init(GPIOC, &gpio_init);

GPIO_PinAFConfig(GPIOC, GPIO_PinSource2, GPIO_AF_SPI2);
GPIO_PinAFConfig(GPIOC, GPIO_PinSource3, GPIO_AF_SPI2);

gpio_init.GPIO_Pin = GPIO_Pin_10;
GPIO_Init(GPIOB, &gpio_init);

GPIO_PinAFConfig(GPIOB, GPIO_PinSource10, GPIO_AF_SPI2);

RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
SPI_I2S_DeInit(SPI1);
spi_init.SPI_Mode = SPI_Mode_Slave;
spi_init.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
spi_init.SPI_DataSize = SPI_DataSize_8b;
spi_init.SPI_CPOL = SPI_CPOL_Low;
spi_init.SPI_CPOL = SPI_CPHA_1Edge;
spi_init.SPI_FirstBit = SPI_FirstBit_MSB;
spi_init.SPI_NSS = SPI_NSS_Soft;
spi_init.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2;
SPI_Init(SPI1, &spi_init);
SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_RXNE, ENABLE);

RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);
SPI_StructInit(&spi_init);
SPI_I2S_DeInit(SPI2);
spi_init.SPI_Mode = SPI_Mode_Master;
spi_init.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
spi_init.SPI_DataSize = SPI_DataSize_8b;
spi_init.SPI_CPOL = SPI_CPOL_Low;
spi_init.SPI_CPOL = SPI_CPHA_1Edge;
spi_init.SPI_FirstBit = SPI_FirstBit_MSB;
spi_init.SPI_NSS = SPI_NSS_Soft;
SPI_Init(SPI2, &spi_init);
```

```
nvic_init.NVIC_IRQChannel = SPI1_IRQn;
nvic_init.NVIC_IRQChannelCmd = ENABLE;
nvic_init.NVIC_IRQChannelPreemptionPriority = 0;
nvic_init.NVIC_IRQChannelSubPriority = 0;
NVIC_Init(&nvic_init);

SPI_Cmd(SPI1, ENABLE);
SPI_Cmd(SPI2, ENABLE);
}

void delay(int count) {
while( - - count);
}
```

5.7. Робота з DMA

Розглянемо приклад програми, яка використовує DMA передавання даних із пам'яті в ЦАП. Зміна рівня сигналу на виході ЦАП відбувається циклічно із частотою зміни в 1 с. Зміни відбуваються на основі значення, яке зчитується з пам'яті сигналу переповнення таймера. В результаті ЦАП відправляє запит до DMA і отримує дані, які записуються в регістр даних, що приводить до зміни рівня сигналу. Схема взаємодії показана на рис.5.1.

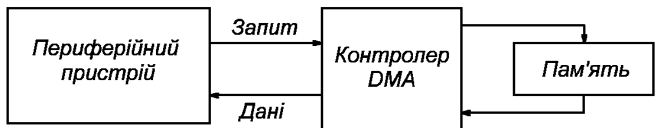


Рис. 5.1. Схема використання DMA

У самій програмі слід звернути увагу на багато параметрів, необхідних для налаштування DMA в порівнянні з іншими пристроями. З цим пов'язана складність використання DMA, оскільки необхідно враховувати велику кількість можливих налаштувань. Проте, багато з них досить прості (напрямок передавання, значення адрес), тому вивчення DMA можна зіставити за складністю з іншими пристроями, в чому допомагає бібліотека Standard Peripheral Library.

```
#include <stm32f4xx.h>
#include <stm32f4xx_rcc.h>
#include <stm32f4xx_dma.h>
#include <stm32f4xx_gpio.h>
```

```
#include <stm32f4xx_tim.h>
#include <stm32f4xx_dac.h>

uint8_t levels[] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77};

void init(void);
void init_gpio(void);
void init_timer(void);
void init_dac(void);
void init_dma(void);

int main(void)
{
init();
while(1)  { }
}

void init(void) {
init_gpio();
init_timer();
init_dac();
init_dma();
}

void init_gpio(void) {
GPIO_InitTypeDef gpio_init;
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
gpio_init.GPIO_Mode = GPIO_Mode_AN;
gpio_init.GPIO_Pin = GPIO_Pin_4;
gpio_init.GPIO_OType = GPIO_OType_PP;
gpio_init.GPIO_PuPd = GPIO_PuPd_NOPULL;
gpio_init.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_Init(GPIOA, &gpio_init);
}

void init_timer(void) {
TIM_TimeBaseInitTypeDef tim_init;
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM6, ENABLE);
tim_init.TIM_CounterMode = TIM_CounterMode_Up;
tim_init.TIM_Period = 16000 - 1;
tim_init.TIM_Prescaler = 1000 - 1;
TIM_TimeBaseInit(TIM6, &tim_init);

TIM_SelectOutputTrigger(TIM6, TIM_TRGOSource_Update);

TIM_Cmd(TIM6, ENABLE);
```

```
}

void init_dac(void) {
DAC_InitTypeDef dac_init;
RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);
DAC_StructInit(&dac_init);
dac_init.DAC_Trigger = DAC_Trigger_T6_TRGO;
dac_init.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
dac_init.DAC_WaveGeneration = DAC_WaveGeneration_None;
DAC_Init(DAC_Channel_1, &dac_init);
}

void init_dma(void) {
DMA_InitTypeDef dma_init;
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1, ENABLE);
DMA_DeInit(DMA1_Stream5);
dma_init.DMA_Channel = DMA_Channel_7;
dma_init.DMA_PeripheralBaseAddr = (uint32_t)(DAC_BASE + 0x10);
dma_init.DMA_Memory0BaseAddr = (uint32_t)&levels;
dma_init.DMA_DIR = DMA_DIR_MemoryToPeripheral;
dma_init.DMA_BufferSize = 8;
dma_init.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
dma_init.DMA_MemoryInc = DMA_MemoryInc_Enable;
dma_init.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
dma_init.DMA_MemoryDataSize = DMA_PeripheralDataSize_Byte;
dma_init.DMA_Mode = DMA_Mode_Circular;
dma_init.DMA_Priority = DMA_Priority_High;
dma_init.DMA_FIFOMode = DMA_FIFOMode_Disable;
dma_init.DMA_FIFOThreshold = DMA_FIFOThreshold_HalfFull;
dma_init.DMA_MemoryBurst = DMA_MemoryBurst_Single;
dma_init.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
DMA_Init(DMA1_Stream5, &dma_init);
DMA_Cmd(DMA1_Stream5, ENABLE);

DAC_Cmd(DAC_Channel_1, ENABLE);
DAC_DMACmd(DAC_Channel_1, ENABLE);
}
```

5.8. Використання аналогового компаратора

Як приклад, розглянемо реалізацію перемикання рівня сигналу з 0 до 1 при перевищенні заданого рівня напруги. Це фактично буде використання простого компаратора. Завдяки тому, що є можливість вибору сигналу для подавання напруги, необхідно подавати сигнал тільки на один із входів

компаратора. В даному випадку як таке джерело обрано значення половини напруги живлення мікроконтролера.

```
#include "stm32f30x_rcc.h"
#include "stm32f30x_gpio.h"
#include "stm32f30x_syscfg.h"
#include "stm32f30x_comp.h"
#include "stm32f30x_misc.h"
#include "stm32f30x_exti.h"

void init(void);

void COMP1_2_3_IRQHandler(void) {
int foo;
uint32_t level;

if (EXTI_GetITStatus(EXTI_Line21) == SET) {
    EXTI_ClearITPendingBit(EXTI_Line21);
    foo = 0;
    level = COMP_GetOutputLevel(COMP_Selection_COMP1);
    if (level == COMP_OutputLevel_High)
    foo = 1;
    }
}

int main() {
init();
while(1) { }
}

void init(void) {
GPIO_InitTypeDef gpio_init;
COMP_InitTypeDef comp_init;
NVIC_InitTypeDef nvic_init;
EXTI_InitTypeDef exti_init;

RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
GPIO_StructInit(&gpio_init);
gpio_init.GPIO_Mode = GPIO_Mode_AN;
gpio_init.GPIO_PuPd = GPIO_PuPd_DOWN;
gpio_init.GPIO_Pin = GPIO_Pin_1;
GPIO_Init(GPIOA, &gpio_init);

gpio_init.GPIO_Mode = GPIO_Mode_AF;
gpio_init.GPIO_Pin = GPIO_Pin_0;
GPIO_Init(GPIOA, &gpio_init);
GPIO_PinAFConfig(GPIOA, GPIO_PinSource0, GPIO_AF_8);
```

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);
COMP_StructInit(&comp_init);
comp_init.COMP_Mode = COMP_Mode_HighSpeed;
comp_init.COMP_OutputPol = COMP_OutputPol_NonInverted;
comp_init.COMP_NonInvertingInput = COMP_NonInvertingInput_IO1;
comp_init.COMP_InvertingInput = COMP_InvertingInput_1_2VREFINT;
COMP_Init(COMP_Selection_COMP1, &comp_init);
COMP_Cmd(COMP_Selection_COMP1, ENABLE);

exti_init.EXTI_Line = EXTI_Line21;
exti_init.EXTI_LineCmd = ENABLE;
exti_init.EXTI_Mode = EXTI_Mode_Interrupt;
exti_init.EXTI_Trigger = EXTI_Trigger_Rising;
EXTI_Init(&exti_init);

nvic_init.NVIC_IRQChannel = COMP1_2_3_IRQn;
nvic_init.NVIC_IRQChannelCmd = ENABLE;
nvic_init.NVIC_IRQChannelSubPriority = 1;
nvic_init.NVIC_IRQChannelPreemptionPriority = 1;
NVIC_Init(&nvic_init);
}
```

Працюючи з компаратором, переконуємося в можливості підключити до входів різні значення сигналів. За аналогією з операційним підсилювачем входи компаратора називаються інвертуючий та неінвертуючий. Переривання від аналогового компаратора у цьому мікроконтролері реалізовані як зовнішні переривання (лінія 21). Тому для їх використання необхідно підключити ті самі модулі, що й для звичайного зовнішнього переривання. Крім того, за роботу компараторів відповідає модуль SYSCFG. Отже, при використанні компараторів подавання тактування на модуль обов'язкове.

5.9. Операційний підсилювач

Ознайомлення з роботою ОП у складі мікроконтролера здійснимо на прикладі, який також використовує ЦАП для генерації напруги, яка за допомогою ОП в режимі підсилювача, що не інвертує, збільшується в 2 рази.

Оскільки використання виводів портів введення/виведення не є додатковою функцією їх роботи, то знайти, які саме виводи будуть використовуватися при роботі ОП, можна скориставшись плагіном MicroXplorer для середовища розробки

Eclipse. У ньому можна вибрати пристрій та дізнатися, як саме використовується виведення мікроконтролера.

Найбільше слід звернути увагу до налаштування ОП. Для цього слід зазначити, які лінії використовуються як входи. У нашому випадку вказуємо, що слід використовувати як неінвертуючий вхід 4 лінію, якій відповідає 1 вивід порту А. На цю лінію подається напруга, що генерується за допомогою ЦАП. Значення напруги на виході 4 порту А – приблизно 0,77 В. З'єднавши за допомогою перемички обидва піни, слід перевірити напругу на виході АЦП – піні 2 порту А. Показники вольтметра становитимуть приблизно 1,55 В. Слід звернути увагу на те, що всі перераховані вище піни налаштовані на роботу в аналоговому режимі.

```
#include "stm32f30x_rcc.h"
#include "stm32f30x_gpio.h"
#include "stm32f30x_opamp.h"
#include "stm32f30x_syscfg.h"
#include "stm32f30x_dac.h"

void init_opamp(void);
void init_dac(void);

int main() {
 init_dac();
 init_opamp();
 while(1);
}

void init_opamp(void) {
 GPIO_InitTypeDef gpio_init;
 OPAMP_InitTypeDef opamp_init;

 RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
 GPIO_StructInit(&gpio_init);
 gpio_init.GPIO_Pin = GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3;
 gpio_init.GPIO_Mode = GPIO_Mode_AN;
 GPIO_Init(GPIOA, &gpio_init);

 RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);
 OPAMP_StructInit(&opamp_init);
 opamp_init.OPAMP_InvertingInput = OPAMP_InvertingInput_PGA;
 opamp_init.OPAMP_NonInvertingInput = OPAMP_NonInvertingInput_IO4;
 OPAMP_Init(OPAMP_Selection_OPAMP1, &opamp_init);
```

```
 OPAMP_PGAConfig(OPAMP_Selection_OPAMP1,
OPAMP_OPAMP_PGAGain_2, OPAMP_PGAConnect_No);
 OPAMP_Cmd(OPAMP_Selection_OPAMP1, ENABLE);
}

void init_dac(void) {
 DAC_InitTypeDef dac_init;
 GPIO_InitTypeDef gpio_init;

 GPIO_StructInit(&gpio_init);
 gpio_init.GPIO_Mode = GPIO_Mode_AN;
 gpio_init.GPIO_Pin = GPIO_Pin_4;
 GPIO_Init(GPIOA, &gpio_init);

 RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);
 DAC_StructInit(&dac_init);
 dac_init.DAC_WaveGeneration = DAC_WaveGeneration_None;
 dac_init.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
 dac_init.DAC_Trigger = DAC_Trigger_None;
 DAC_Init(DAC_Channel_1, &dac_init);

 DAC_Cmd(DAC_Channel_1, ENABLE);
 DAC_SetChannel1Data(DAC_Align_8b_R, 0x44);
}
```

Слід звернути увагу, що зазначена функціональність реалізована за допомогою простої ініціалізації пристроїв у складі мікроконтролера. Серед цих пристроїв є також ЦАП. Важливо пам'ятати, як саме реалізується подача необхідної розрахункової напруги ЦАП, встановлюються значення в регістрі даних, що перетворюється на потрібний рівень напруги. Значення вихідної напруги на виході ЦАП визначається з урахуванням напруги живлення.

5.10. Література до розділу

1. Мікроконтролери та мікропроцесорна техніка. Лабораторний практикум. [Електронний ресурс] : навч. посіб. для студ. спеціальності 151 «Автоматизація та комп’ютерно-інтегровані технології» / О.М. Павловський; КПІ ім. Ігоря Сікорського, 2021. — 104 с.
2. Програмування вбудованих систем : метод. вказівки до виконання лабораторних робіт для студентів денної та заочної форми навчання за спеціальністю 123 “Комп’ютерна інженерія ” /

уклад. Дрєєва Г.М., Дрєєв О.М., Денисенко О.О., Коноплицька-Слободенюк О.К. — Кропивницький: ЦНТУ, 2018. — 90 с.
3. Лабораторный практикум для изучения микроконтролеров архитектуры ARM Cortex-M4 на базе отладочного модуля STM32F4 Discovery / Бугаев В.И., Мусиенко М.П., Крайнык Я.М. – Москва-Николаев: МФТИ-ЧГУ, 2013. – 71 с.
4. Лабораторный практикум по изучению микроконтролеров STM32 на базе отладочного модуля STM32F3 Discovery / Бугаев В.И., Мусиенко М.П., Крайнык Я.М. – Москва-Николаев: МФТИ-ЧГУ, 2014. – 33 с.

ДОДАТКИ

ДОДАТОК А

КЛАСИФІКАЦІЯ ТА МАРКУВАННЯ МІКРОКОНТРОЛЕРІВ STM

ДА.1 Класифікація мікроконтролерів STM

Мікроконтролери STM мають багато серій, які можуть задовольнити різні потреби ринку:

CPU	Ядро	Серія	Опис
32	Cortex-M0	STM32-M0	Початковий рівень
		STM32-L0	Низьке енергоспоживання
	Cortex-M3	**STM32-F1**	**Основний тип, основна частота 72 МГц**
		STM32-F2	Висока продуктивність
		STM32-L1	Низьке енергоспоживання
	Cortex-M4	STM32-F3	Змішаний сигнал
		STM32-F4	**Висока продуктивність, основна частота 180 МГц**
		STM32-L4	Низьке енергоспоживання
	Cortex-M7	STM32-F7	Висока продуктивність
8	6502 (супер-видання)	**STM8S**	**Стандартні серії**
		STM8AF	Стандартна серія автомобільних застосунків
		STM8AL	Автомобільні застосунки малої потужності
		STM8L	Низьке енергоспоживання

Отже, сімейство мікроконтролерів STM32, заснованих на 32-бітних ядрах ARM Cortex, складається з 16 серій мікроконтролерів: F0, F1, F2, F3, F4, F7, L0, L1, L4, L4+, L5, G0, G4, H7, WB, WL:

Серія STM32	Ядро ARM CPU
L5	Cortex-M33
F7, H7	Cortex-M7F
F4, F3, L4, J	Cortex-M4F
F2, F1, L1, W, J	Cortex-M3
L0, J	Cortex-M0+
F0, J	Cortex-M0

Кожна серія базується на одному із ядер ARM: Cortex-M33, Cortex-M7F, Cortex-M4F, Cortex-M3, Cortex-M0+, Cortex-M0. Виробник поділяє всі серії мікроконтролерів STM32 на 4 платформи (групи):

Платформа мікроконтролерів	**Назви серій, що входять в групу**
Високопродуктивні	F2, F4, F7, H7
Широкого застосування	F0, G0, F1, F3, G4
Наднизького споживання	L0, L1, L4, L4+, L5
Безпровідні	WB, WL

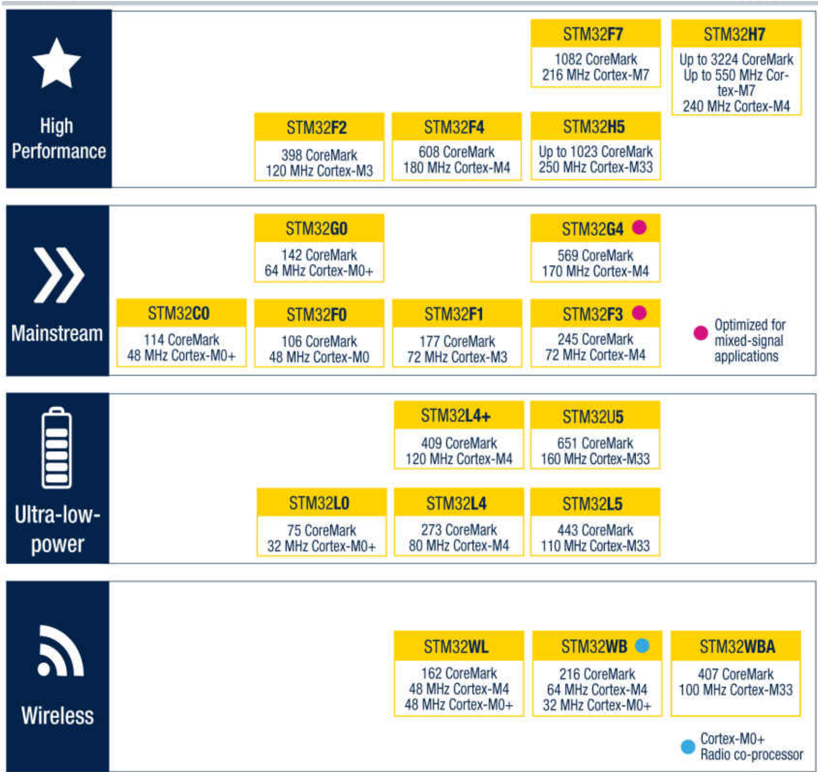
Мікроконтролери STM32 поділяють на лінії:

- **Value line** STM32F100xx – 24 МГц CPU;
- **Access line** STM32F101xx – 36 МГц CPU, до 1 Мб Flash-пам’яті;
- **USB access line** STM32F102xx – 48 МГц CPU, містять USB FS;
- **Performance line** STM32F103xx – 72 МГц, до 1 Мб Flash-пам’яті, USB і CAN;
- **Connectivity line** STM32F105/107xx – 72 МГц CPU, містять Ethernet MAC, CAN і USB 2.0 OTG.

За розміром пам’яті програм поділяються на:

- Пристрої з низькою щільністю (**Low-density devices**) STM32F101xx, STM32F102xx і STM32F103xx, де розмір флеш-пам’яті коливається від 16 до 32 Кбайт.
- Пристрої середньої щільності (**Medium-density devices**) STM32F101xx, STM32F102xx і STM32F103xx, де розмір флеш-пам’яті коливається від 64 до 128 Кбайт.
- Пристрої високої щільності (**High-density devices**) STM32F101xx і STM32F103xx, де розмір флеш-пам’яті коливається від 256 до 512 Кбайт.
- Пристрої комунікаційної лінії (**Connectivity line devices**) STM32F105xx і STM32F107xx.

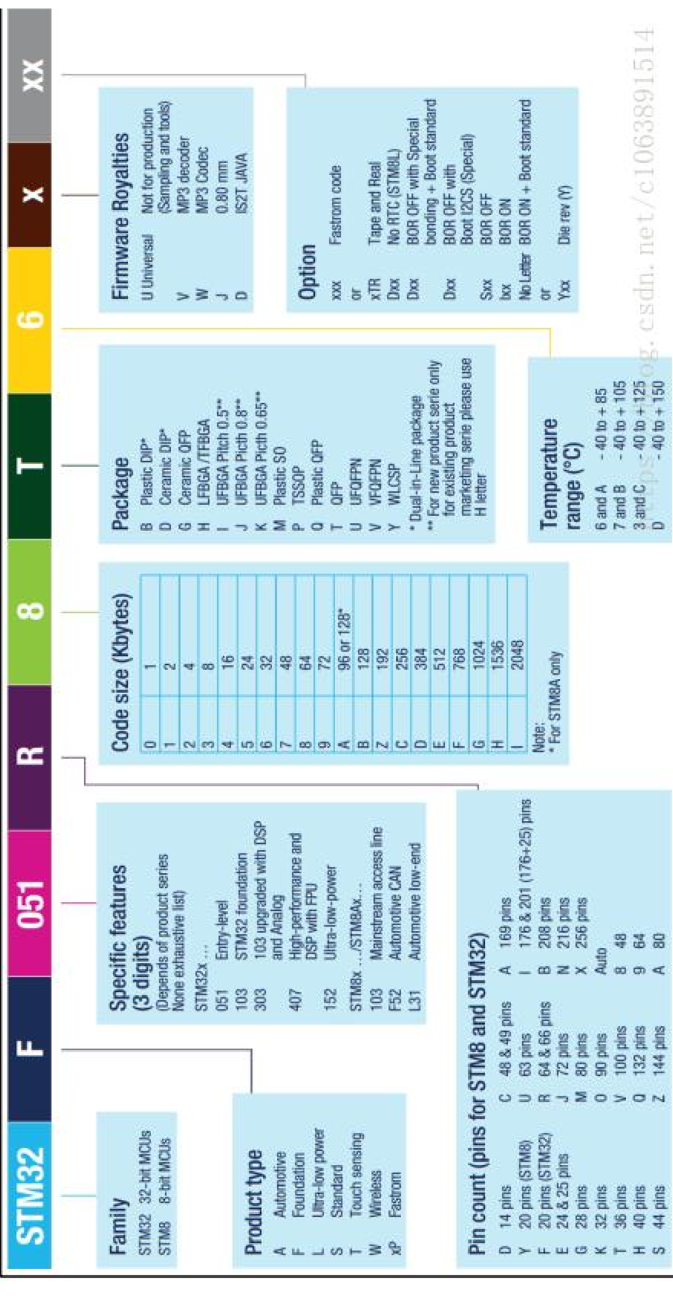
Області застосування мікро STM32 контролерів показані на рисунку нижче:



ДА.2 Маркування мікроконтролерів STM

Назва кожного 32-бітного мікроконтролера (MCU) починається з STM32, де STM означає STMicroelectronics, 32 – 32-бітний мікроконтролер. Далі іде велика літера, яка описує тип продукту. Наступна цифра вказує, яке ядро ARM Core використовувалося. Наступні дві цифри вказують лінію MCU, особливості пристрою. Наступна літера позначає кількість ніжок. Після цього вказується цифра або літера, що позначає розмір флеш-пам’яті MCU. Наступна буква вказує тип корпусу. Остання цифра інформує про діапазон робочих температур пристрою.

Можуть далі наводитися й додаткові буквені позначення, які вказують на використовувані ліцензії в мікроконтролері, опціональні апаратні налаштування.


STM32
F
051
R
8
T
6
x
xx
Family
STM32 32-bit MCUs
STM8 8-bit MCUs
Product type
A Automotive
F Foundation
L Ultra-low power
S Standard
T Touch sensing
W Wireless
xP Fastrom
Specific features (3 digits)
(Depends of product series None exhaustive list)
STM32x ...
051 Entry-level
103 STM32 foundation
303 103 upgraded with DSP and Analog
407 High-performance and DSP with FPU
152 Ultra-low-power
STM8x .../STM8Ax...
103 Mainstream access line
F52 Automotive CAN
L31 Automotive low-end
Pin count (pins for STM8 and STM32)
D 14 pins
Y 20 pins (STM8)
F 20 pins (STM32)
E 24 & 25 pins
G 28 pins
K 32 pins
T 36 pins
H 40 pins
S 44 pins
C 48 & 49 pins
U 63 pins
R 64 & 66 pins
J 72 pins
M 80 pins
O 90 pins
V 100 pins
Q 132 pins
Z 144 pins
A 169 pins
I 176 & 201 (176+25) pins
B 208 pins
N 216 pins
X 256 pins
Auto
8 48
9 64
A 80
Code size (Kbytes)
0 1
1 2
2 4
3 8
4 16
5 24
6 32
7 48
8 64
9 72
A 96 or 128*
B 128
Z 192
C 256
D 384
E 512
F 768
G 1024
H 1536
I 2048
Note:
* For STM8A only
Package
B Plastic DIP*
D Ceramic DIP*
G Ceramic QFP
H LFBGA /TFBGA
I UFBGA Pitch 0.5**
J UFBGA Picth 0.8**
K UFBGA Picth 0.65**
M Plastic SO
P TSSOP
Q Plastic QFP
T QFP
U UFQFPN
V VFQFPN
Y WLCSP
* Dual-in-Line package
** For new product serie only for existing product marketing serie please use H letter
Temperature range (°C)
6 and A - 40 to + 85
7 and B - 40 to + 105
3 and C - 40 to + 125
D - 40 to + 150
Firmware Royalties
U Universal Not for production (Sampling and tools)
V MP3 decoder
W MP3 Codec
J 0.80 mm
D IS2T JAVA
Option
xxx Fastrom code
or
xTR Tape and Real
Dxx No RTC (STM8L)
Dxx BOR OFF with Special bonding + Boot standard
Dxx BOR OFF with Boot I2CS (Special)
Sxx BOR OFF
Ixx BOR ON
No Letter BOR ON + Boot standard
or
Yxx Die rev (Y)

Елементи маркування:

1) фірма виробник – STMicroelectronics;

2) розрядність мікроконтролера, біт – 8 або 32;

3) тип продукту.

A	Automotive (Автомобільний)
F	Foundation (Основний) High-perfomance (STM32 F2 series)
G	Mainstream
L	Ultra-low power (Наднизька споживана потужність)
H	High-perfomance (Високопродуктивний)
S	Standart (Стандарт)
T	Touch sensing (Розпізнавання дотику)
W	Wireless (Бездротовий)
xP	Fast ROM (Швидке ПЗП)

Цікаво відзначити, що назви здебільшого відповідають схемі, однак вони розвиваються разом із продуктами, і деякі назви випадають зі схеми. Наприклад, серія STM32 F2, яка є високопродуктивним мікроконтролером.

4) Специфічні особливості

Ядро:

0	ARM Cortex M0
1	ARM Cortex M3
2	ARM Cortex M3
3	ARM Cortex M4
4	ARM Cortex M4
7	ARM Cortex M7

5) Число ніжок

D	14		O	90
Y	20 (STM8)		V	100
F	20 (STM32)		Q	132
E	24 або 25		Z	144
G	28		A	169
K	32		I	176 або 201
T	36		B	208
H	40		N	216
S	44		X	256

C	48 або 49		Auto	
U	63		8	48
R	64 або 66		9	64
J	72		A	80
M	80			

6) Об’єм пам’яті програм, Кбайт

0	1		A	96 (128 для STM8A)
1	2		B	128
2	4		Z	192
3	8		C	256
4	16		D	384
5	24		E	512
6	32		F	768
7	48		G	1024
8	64		H	1536
9	72		I	2048

7) Корпус мікросхеми мікроконтролера

B	Пластиковий DIP
D	Керамічний DIP
G	Керамічний QFP
H	LFBGA / TFBGA
I	UFBGA крок 0,5 для нових серійних продуктів
J	UFBGA крок 0,8 для нових серійних продуктів
K	UFBGA крок 0,65 для нових серійних продуктів
M	Пластиковий SO
P	TSSOP
Q	Пластиковий QFP
T	QFP
U	UFQFPN
V	VFQFPN
Y	WLCSP

8) Діапазон температур, °C

6 або A	-40…+85
7 або B	-40…+105
3 або C	-40…+125
D	-40…+150

9) Використовувані ліцензії

U	Універсальні. Не для виробництва (Вибірки та інструменти)
V	MP3 декодер
W	MP3 кодек
J	0,8 мм
D	1S2T JAVA

10) Опціональні апаратні налаштування.

xxx	FastROM код
xTR	стрічка або котушка
Dxx	без RTC (для STM8L)
Dxx	BOR OFF зі спеціальним з’єднанням + стандартне завантаження
Dxx	BOR OFF із завантаженням I2CS (спеціальні)
Sxx	BOR OFF
lxx	BOR ON
___ або Yxx	BOR ON+ стандартне завантаження

Майже всі моделі сімейства мікроконтролерів STM32 будуть дотримуватися цих правил маркування. Однак у деяких рідкісних випадках можуть бути невідповідності. Щоб розшифрувати ідентифікатор MCU, розбийте його назву на описані частини та використовуйте таблиці для розшифровки значень.

Приклад розшифрування. Нехай маємо мікроконтролер **STM32L151R8T6**:

1) **STM** - STMicroelectronics,
2) **32** – 32-бітний,
3) **L** - наднизька споживана потужність,
4) **1** - мікропроцесорне ядро Cortex M3,
5) **51** - модель 51,
6) **R** - 64-ніжки,
7) **8** - 64 Кбайти флеш-пам’яті,
8) **T** - корпус LQFP,
9) **6** - діапазон температур роботи від -40 °С до 85 °С.

Література до додатка А

https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html
https://stm32world.com/wiki/STM32
https://www.pianshen.com/article/7566833649/
https://arm-stm.blogspot.com/2013/07/stm32f10x.html
https://russianblogs.com/article/9357272733/
https://www.digikey.com/en/maker/blogs/2020/understanding-stm32-naming-conventions
https://wikipedia.org/wiki/STM32

ДОДАТОК Б

ЕТАПИ ПРОГРАМУВАННЯ МІКРОКОНТРОЛЕРІВ

Можна виділити такі етапи програмування мікроконтролера:

1. Постановка задачі.
2. Проєктування схеми структурної пристрою на основі мікроконтролера.
3. Розробка схеми принципової пристрою.
4. Розробка алгоритму роботи мікроконтролера.
5. Перевірка відповідності алгоритму схемі принциповій.
6. Корекція принципової схеми.
7. Написання власне керуючої програми мікроконтролера.
8. Діагностика та налагодження програми за допомогою програм налагодження (MpLab, Proteus, та ін.).
9. Випробування пристрою на макеті.
10. За результатами випробувань можна повернутись на будь-який з пунктів 1-8 або рекомендувати пристрій для серійного виготовлення.

На етапі постановки задачі необхідно визначитися, які функції виконує пристрій в цілому і мікроконтролер зокрема. На основі цього можна скомпонувати структурну схему пристрою.

Принципова схема пристрою проєктується згідно з наявною елементною базою та рівнем підготовки виконавця. Але в будь-якому випадку схемотехнічне рішення може бути двох типів: ефективне, та неефективне. Може також знадобитися корекція структурної схеми пристрою. Визначальним фактором на даному етапі є співвідношення ціна/функціональність. На цьому етапі остаточно визначаються з вибором найефективнішого мікроконтролера для конкретного конструкторського рішення.

На основі отриманої принципової схеми та постановки задачі можна скласти алгоритм роботи програми керування пристрою. Після наявності схеми принципової та алгоритму роботи мікроконтролера приступають до написання керуючої програми мікроконтролера. При написанні програми програміст повинен користуватися такими документами:

Обов’язково:

1. Таблиця команд даного типу мікроконтролера.
2. Карта пам’яті мікроконтролера.

Необов’язково, але дуже бажано:

1. Оригінальна фірмова документація на мікроконтролер.

Для написання програми можна використовувати програми налагодження та симуляції. Це значно скорочує час написання програми та зменшує кількість помилок в ній.

Навчальне видання

МІКРОПРОЦЕСОРИ В РАДІОСИСТЕМАХ І ПРИСТРОЯХ: ПОЧАТОК РОБОТИ З МІКРОКОНТРОЛЕРАМИ STM32

Навчальний посібник

Укладачі: **ВЕРИГА** Андрій Дмитрович,
ПОЛІТАНСЬКИЙ Руслан Леонідович,
КРУЛІКОВСЬКИЙ Олег Валерійович,
КОСОВАН Григорій Васильович

Відповідальний за випуск Шпатар П.М.

Літературний редактор Колодій О.В.

Підписано до друку 01.12.2023. Формат 60х84/16.
Електронне видання
Ум.- друк.арк. 10,3. Обл.-вид.арк. 11,1. Зам. Н-086.

Видавництво та друкарня Чернівецького національного університету імені Юрія Федьковича
58002, Чернівці, вул. Коцюбинського,2
e-mail: ruta@chnu.edu.ua
Свідоцтво суб'єкта видавничої справи ДК №891 від 08.04.2002