МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЦЕНТРАЛЬНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

Програмування вбудованих систем

Методичні вказівки до виконання лабораторних робіт
для студентів денної та заочної форми навчання за спеціальністю
123 “Комп’ютерна інженерія ”

ЗАТВЕРДЖЕНО
на засіданні кафедри кібербезпеки та
програмного забезпечення,
протокол від 28 квітня 2018 року № 14

КРОПИВНИЦЬКИЙ
2018

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.

Укладачі: Дрєєва Г.М., Дрєєв О.М. , Денисенко О.О.

Рецензенти: Якименко М.С. – кандидат фізико-математичних наук, доцент кафедри кібербезпеки та програмного забезпечення.

УДК 378.147:317

Рекомендовано до друку на засіданні кафедри кібербезпеки та програмного забезпечення (протокол № 14 від 28.04.2018 р.). Рецензенти: Якименко М.С. – кандидат фізико-математичних наук, доцент кафедри кібербезпеки та програмного забезпечення.

Програмування вбудованих систем: методичні вказівки для студентів денної та заочної форми навчання за спеціальністю 123 “Комп’ютерна інженерія”, 125 “Кібербезпека” / уклад. Дрєєва Г.М., Дрєєв О.М., Денисенко О.О., Коноплицька-Слободенюк О.К. — Кропивницький: ЦНТУ, 2017. — 90 с.

Дисципліна є вибірковою у підготовці фахівців з системного програмування. Дисципліна надає знання в області структури засобів створення програмного забезпечення для вбудованих систем, які характеризуються вимогами до обчислювальних систем такі як: автономність, енергоефективність, малогабаритність, надійність та подібні. Для великої частини випадків, виконання зазначених умов можливе при використанні енергоефективних мікроконтролерів з малою кількістю оперативної пам’яті, що приводить до неможливості використання повноцінних операційних систем. Тому на програміста покладено завдання розподілу часу між задачами та використання ефективних, з точки зору використання пам’яті мікроконтролеру, алгоритмів. На відміну від програмування на апаратному рівні, для сучасних мікроконтролерів є можливість використовувати фреймворки, де необхідність керувати периферією за допомогою регістрів відсутня – програмісту надано структури даних з налаштуваннями та функції для їх застосування, що значно розширює універсальність твореного програмного забезпечення. Всі навчальні експерименти поставлено для ARM мікроконтролерів з орієнтацією на плату проектування STM32F4Discovery.

В результаті вивчення дисципліни студенти повинні знати:

– основні архітектури мікроконтролерних програм;

– основні методи таймінгу;

– основні методи формування дискретних та аналогових сигналів;

– технології та програмні засоби розробки програм для вбудованих систем на мові програмування С.

В результаті вивчення дисципліни студенти повинні вміти:

– створювати власний код програмного забезпечення для мікроконтролерів вбудованих систем;

– керувати світлодіодами, двигунами постійного струму, кроковими двигунами;

– використовувати рідкокристалеві дисплеї для виводу інформації;

– використовувати зовнішні пристрої за допомогою стандартних інтерфейсів;

– виконувати задачі за принципами розділення процесорного часу між задачами.

Зміст

ВСТУП

Матеріал лабораторного практикуму служить для допомоги у вивченні можливостей 32-розрядних мікроконтролерів і їх використань в побудові інформаційних систем, організації взаємодії декількох пристроїв між собою, використання передачі і відображення інформації в комп'ютерних системах, навчанні самостійної розробки програмного забезпечення з використанням спеціалізованих програм, проведенні тестування та налагодження на реальних пристроях.

В ході виконання лабораторних робіт вивчаються принципи роботи мікроконтролерів, їх основною периферії, організації передачі даних з їх використанням, управління іншими пристроями для виміру зовнішніх показників, налаштувати зовнішній вигляд інформації, отриманої від зовнішніх пристроїв.

Навчають отримують можливість на практиці самостійно налаштувати роботу демонстраційних прикладів і розробити власні відповідно до індивідуального завданням.

Матеріал лабораторного практикуму складається з трьох основних розділів:

- загальний опис архітектури ARM і 32-розрядних мікроконтролерів STM;

- загальна інформація, яка необхідна для початку роботи з налагоджування платою STM32F4Discovery;

- вісім лабораторних робіт для вивчення основних можливостей, пристроїв і характеристик плати:

ШІМ, АЦП, USART, SPI, DMA, таймери та інше.

Крім того, в додатку наводиться інформація щодо можливості підключення до плати дисплея.

Для виконання лабораторних робіт переважно знання основ теорії цифрової схемотехніки, розробки програмного забезпечення та алгоритмізації, а також знання мови програмування C ++

1. ОПИС АРХІТЕКТУРИ ARM і 32-розрядні МИКРОКОНТРОЛЛЕРИ STM

1.1. 32-х розрядна архітектура ARM

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

Хоча ARM засновані 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:

1. Змінна кількість циклів виконання для простих інструкцій. Прості інструкції ARM можуть виконуватися декілька циклів. Наприклад, виконання інструкцій Load і Save залежить від кількості регістрів, які їм передані.

2. Можливість поєднувати команди зсуву і обертання з командами обробки інформації.

3. Умовне виконання - інструкція виконується лише в тому випадку, якщо виконується конкретна умова. Це збільшує продуктивність і дозволяє позбутися від операторів розгалуження.

4. Поліпшені інструкції - процесори ARM підтримують поліпшені DSP-інструкції для операцій з цифровими сигналами.

Програміст може розглядати ядро ARM як набір функціональних блоків - ALU, MMU і 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.

Процесори ARM містять до 18 регістрів: 16 регістрів даних і 2 регістра процесів. Усі регістри містять 32 біта і іменуються від R0 до R15. Регістри R13, R14, R15 використовуються для виконання певних специфічних завдань:

a) R13 використовується в якості покажчика стека;

b) R14 використовується як зв'язуючий регістр;

c) R15 грає роль лічильника.

Залежно від контексту ці регістри можуть використовуватися як регістри загального призначення. Також є два програмних регістра, які називаються CPSR (Current Program Status Register) і SPSR (Saved Program Status Register), які використовуються для збереження стану процесора і програми.

Рис. 1.1. Складові модулі процесорів ARM Cortex-M4

Одними з останніх процесорів для вбудованих систем, є процесори, засновані на архітектурі ARM Cortex-M4. Ці процесори призначені для використання в цифровій обробці сигналів (Digital Signal Processing, DSP). У загальному вигляді мікроконтролери, засновані на базі ARM Cortex-M4 мають такі внутрішні модулі (рис. 1.1): Мікроконтролер, встановлений на розглянутій платі, STM32F407VG, в якості основи використовує саме рішення ARM Cortex-M4.

1.2. Особливості будови STM32 процесорів

Мікроконтролери STM32 побудовано з використанням 32-розрядний ядра Cortex різних версій (в мікроконтролері, встановленому на платі stm32f4Discovery використовується ядро Cortex-M4). Основні характеристики ядра мікроконтролерів STM32 представлені в таблиці 1.1.

Таблиця 1.1.

Основні характеристики ядра мікроконтролерів STM32

Характеристика	Пояснення
Ширина слів для даних, розрядів	32
Архітектура	Гарвард
Конвеєр виконання інструкцій	3-х ступеневий
Набір інструкцій	RISC
Розрядність пам'яті програм	32
Буфер попередньої вибірки, розрядів	2x64
Середній розмір інструкцій, байтів	2
Тип переривань	Векторизовані
Затримка реакції на переривання	12 циклів
Режими керування енергозабезпеченням	Сон, сон по виходу, глибокий сон
Інтерфейс програмування та налагодження	ST-LINK, JTAG

Мікроконтролери STM32Fxxx типу побудовані на гарвардської архітектурі і мають 3-х ступеневий конвеєр, який зменшує час виконання команд. Вони

розроблені для побудови систем з максимальною енергоефективністю і мають кілька режимів управління енергоспоживанням. Також використовуються внутрішні інтерфейси пам'яті шириною більше, ніж середня довжина інструкції. Це мінімізує число звернень до шини пам'яті, тому споживання електроенергії, пов'язане з операціями по шині і читанням незалежній пам'яті є мінімальним. Використана технологія обробки переривань з виключенням внутрішніх операцій над стеком (tail chaining), скорочує час реакції на переривання і зменшує кількість необхідних операцій зі стеком.

На рис. 1.2 представлено спрощене уявлення цифрового периферійного пристрою.

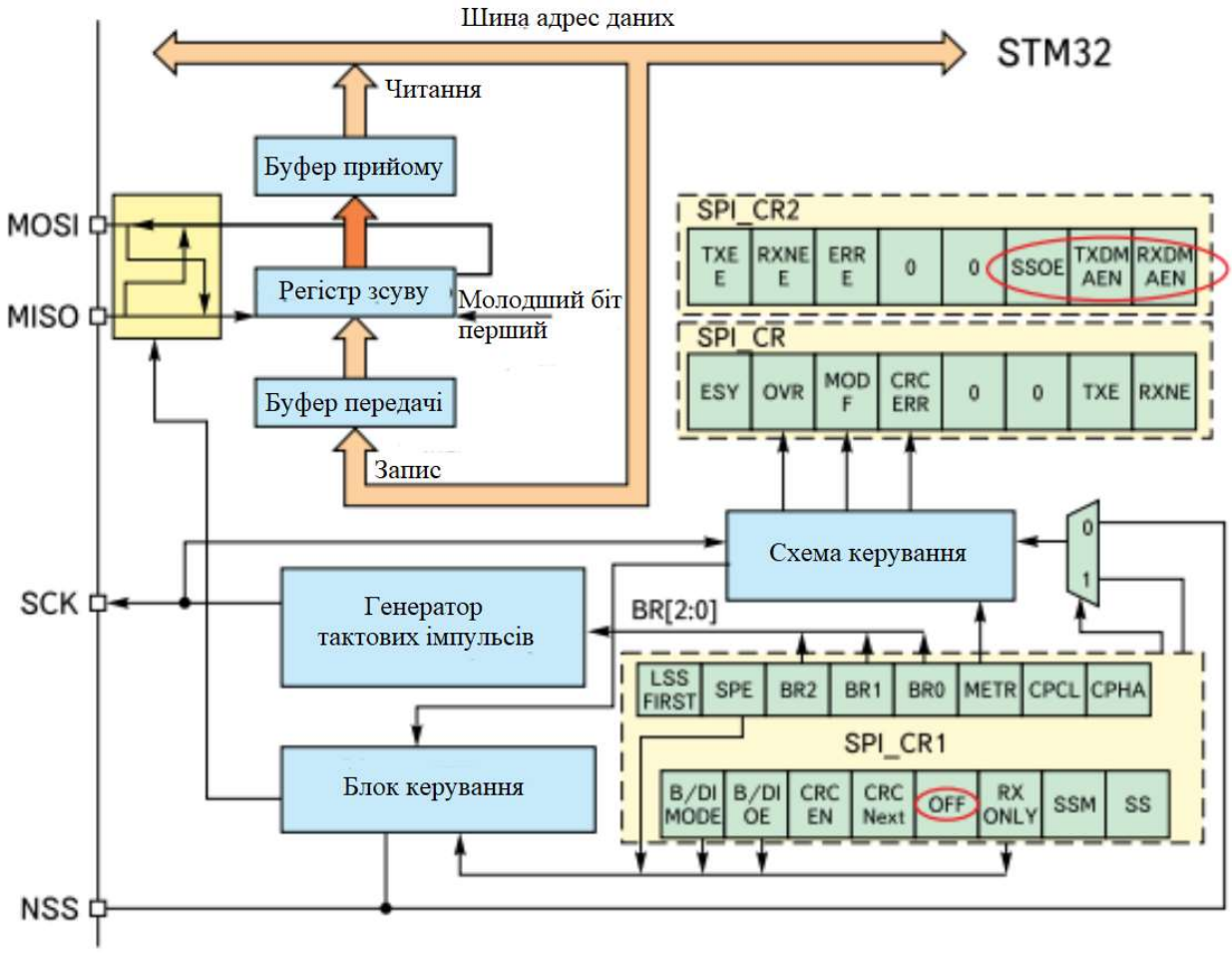


Рис. 1.2. Структура периферійного пристрою

Периферійний вузол може бути розділений на два основні блоки. Перший

блок - це ядро, яке має архітектуру на скінчених автоматах, лічильники і будь-який вид комбінаторної або послідовної логіки. Вони призначені для виконання завдань, які потребують участі процесора, таких як прості завдання передачі даних, управління аналоговими входами або виконання функцій, прив'язаних до синхросигналів. Ядро периферійного вузла зв'язується із зовнішнім світом через порти введення/виводу МК. Зовнішні з'єднання можуть складатися з декількох сигналів або складних шин. Другий блок - налаштування і управління периферією, які здійснюються програмою через регістри, з'єднані з внутрішньою шиною, яка розділяється з іншими ресурсами МК.

1.3. Короткий опис STM32F4 Discovery

Плата STM32F4 Discovery (рис. 1.3) призначена для ознайомлення з можливостями 32-х розрядного МК на основі ARM-архітектури, а також для реалізації власних пристроїв і програм з використанням апаратного забезпечення платою.

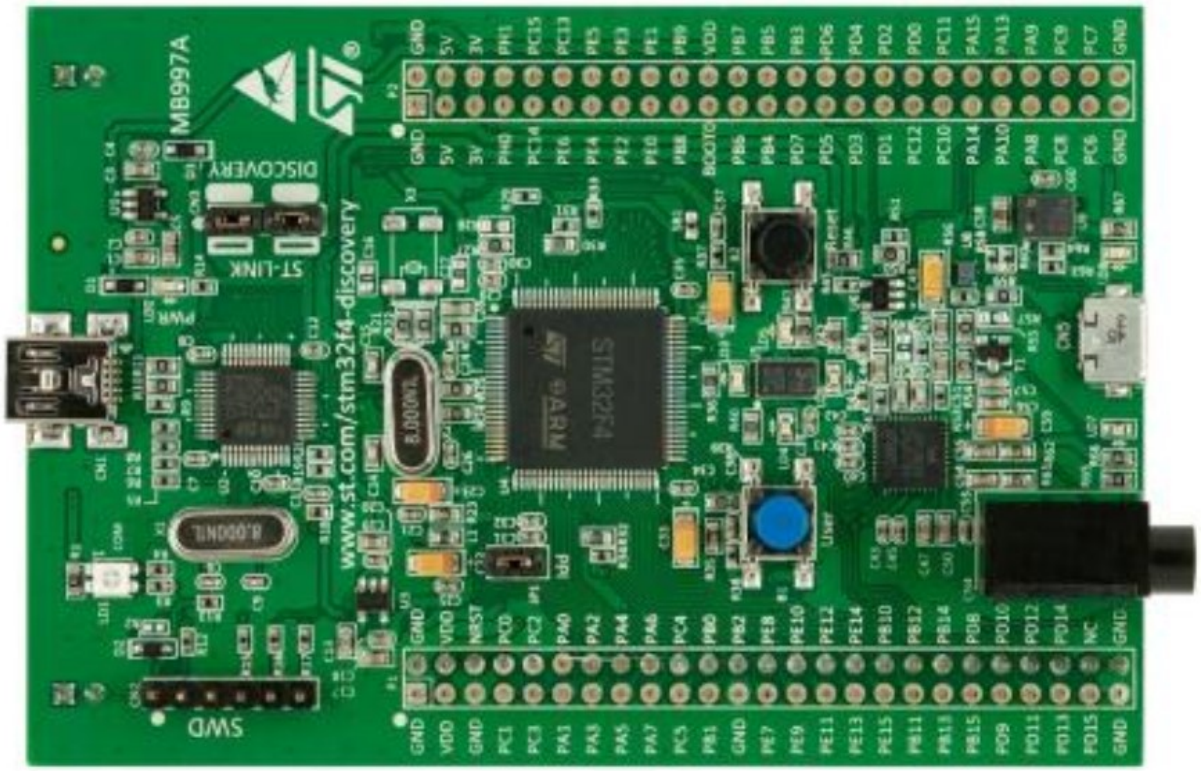


Рис. 1.3. Зовнішній вигляд плати STM32F4Discovery

Плата STM32F4Discovery забезпечена наступними пристроями:

- мікроконтролером STM32F407VGT6 з ядром Cortex-M4F тактовою частотою 168 МГц, 1 Мб;
- Flash-пам'яті, 192 кб RAM в корпусі LQFP100;
- налагоджувачем ST-Link / V2 для налагодження та програмування МК;
- живленням плати через USB або від зовнішнього джерела живлення 5В;
- датчиком руху ST MEMS LIS302DL і виходами цифрового акселерометра по трьох осях;
- датчиком звуку ST MEMS MP45DT02;
- звуковим ЦАП CS43L22;
- вісьмома світлодіодами: LD1 (червоний / зелений) для USB-підключення, LD2 (червоний) для живлення 3.3 В, чотири призначені для користувача світлодіода: LD3 (рожевий), LD4 (зелений), LD5 (червоний), LD6 (синій) і два світлодіода для USB LD7 (зелений) і LD8 (червоний);
- - двома кнопками (для програмування користувачем і для перезапуску).

Завдяки наявності мінімального набору периферійних пристроїв, плата може бути використана для створення багатьох проектів без додаткових пристроїв, та в разі потреби їх можна приєднати за допомогою виведених контактних площадок-штекерів. Зручним є те, що кожен з контактів є підписаним, тому в більшості випадків потреба у зверненні до принципової схеми плати не є обов'язковою.

2. ПОЧАТОК РОБОТИ З STM32F4Discovery

Плата розробника під'єднується до ПК за допомогою кабелю USB з роз'ємами USB типу A та mini-USB типу B. У комплекті такий шнур не надається, тому його потрібно придбати окремо. Зовнішній вигляд конекторів кабелю показано на рис. 2.1.

Рис. 2.1. Конектори кабелю для приєднання STM32F4Discovery до ПК

При підключенні будьте уважні: на платі є два роз'єми USB, один з яких призначений для підключення, а другий - для реалізації роботи з USB.

Передбачається, що на комп'ютері встановлена операційна система сімейства Windows (XP, або новіші). Існує можливість підключення до ПК з Linux, але це зажадає набагато більше зусиль по встановленню програмного забезпечення, бо офіційно засобів програмування STM32 під Linux немає, а відкриті проекти вимагають значної кількості погано задокументованих налаштувань та “ручної” роботи.

2.1. Середовище розробки Keil uVision

Для початку роботи слід встановити середовище розробки. Далі розглянуто процес установки і початку роботи для одного з найпопулярніших засобів розробки для ARM - Keil. Розглянемо деталі установки та використання середовища розробки Keil uVision.

Завантажити програму можна з сайту виробника — https://www.keil.com/download/product/, вибравши для завантаження пункт MDK-ARM останньої версії, після чого потрапляємо на сторінку, на якій потрібно ввести свої персональні дані (рис 2.2).

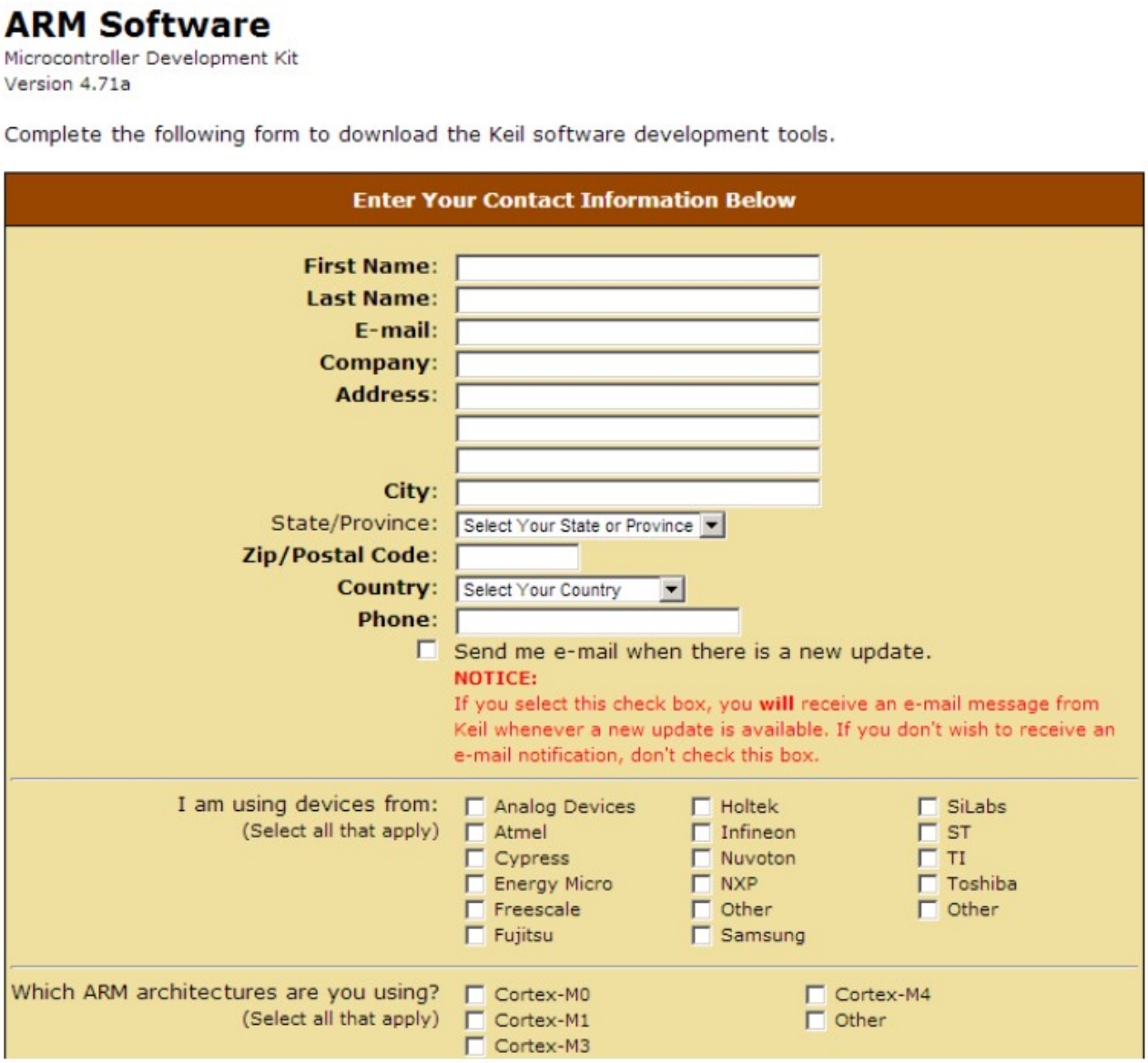


Рис. 2.2. Реєстраційна форма для завантаження Keil uVision

Після заповнення форми надається можливість завантажити програмний продукт. Інсталяція потребує досить багато вільного дискового простору (близько 2.5 Гб), тому слід заздалегідь підготувати необхідні ресурси. Після запуску установки відкриється наступне вікно (рис. 2.3).

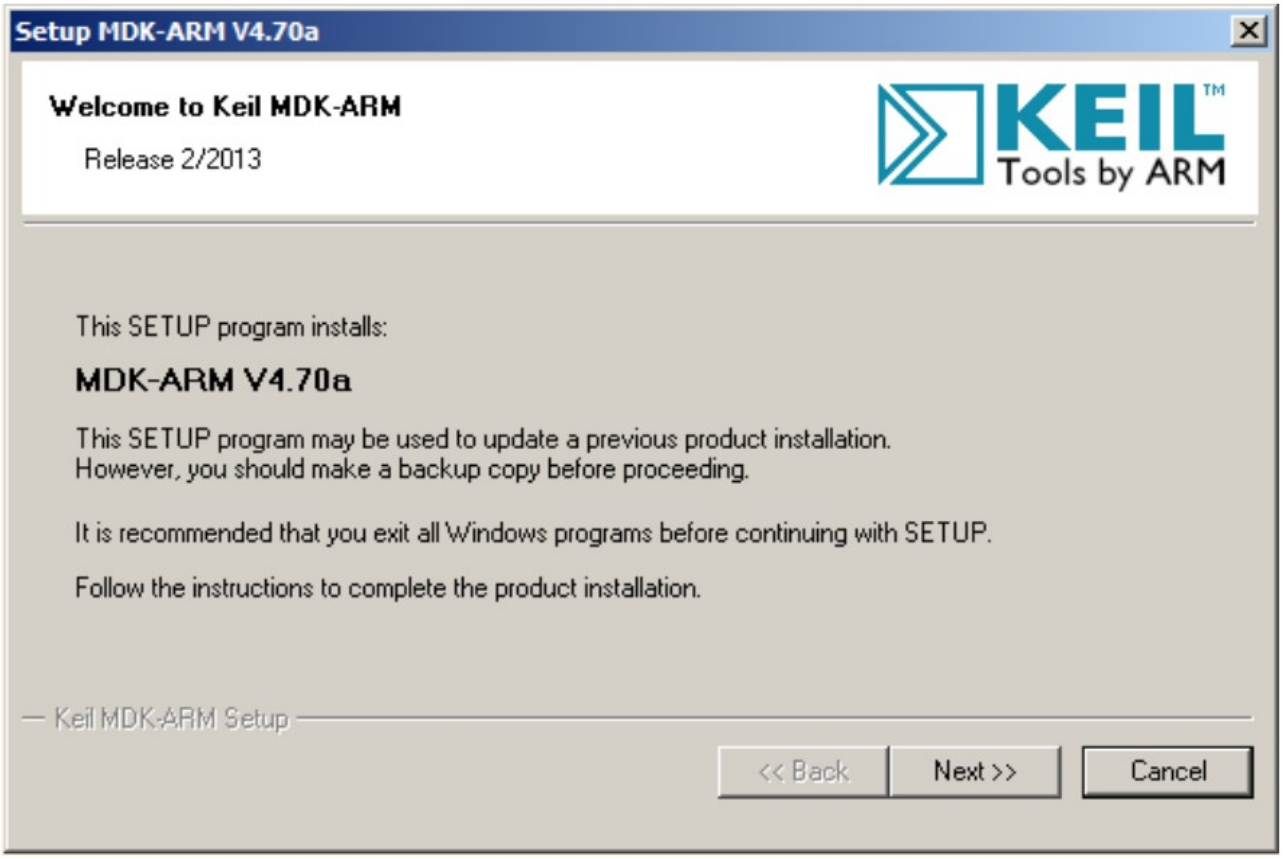


Рис. 2.3. Початок інсталювання Keil

Далі необхідно прийняти умови ліцензійної угоди, а також вибрати директорію для інсталяції (рис. 2.4), потім потрібно вказати інформацію про користувача.

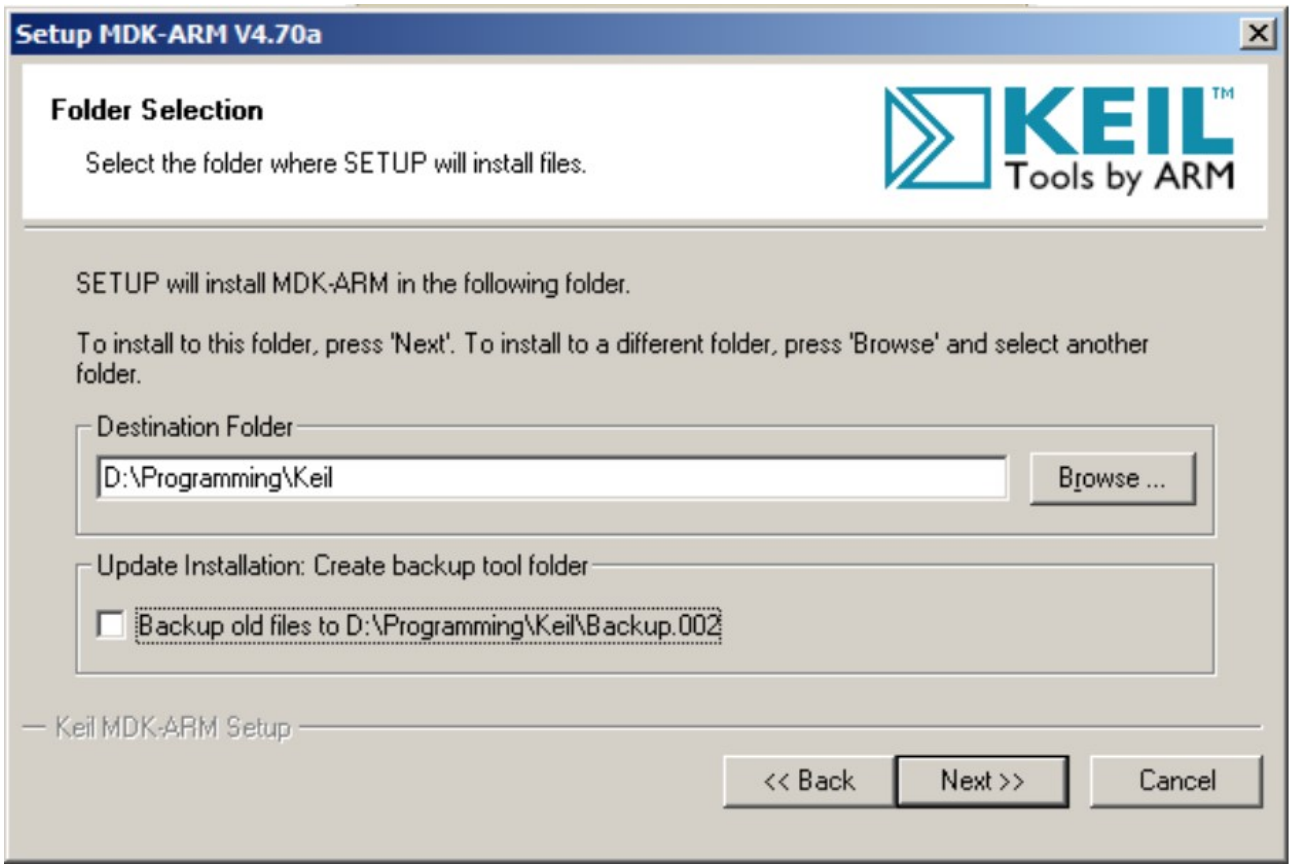


Рис. 2.4. Вибір директорії для інсталювання

Після цього почнеться безпосередньо процес інсталювання.

Крім самого середовища розробки для написання програм необхідні також бібліотеки Cortex Microcontroller Software Interface Standart (CMSIS) і Standart Peripheral Library (SPL). Краще за все їх завантажити з сайту виробника st.com.

Бібліотека SPL служить для управління всіма основними пристроями, що входять до складу мікроконтролера: USART, SPI, DMA, ADC, DAC, ... Основною перевагою бібліотеки є те, що вона робить більш зрозумілим управління мікроконтролером для розробника, в тому числі і початківця, який ще не знає всіх тонкощів налаштувань напряму через регістри. Крім того, разом із самою бібліотекою поширюються і демонстраційні проекти для роботи з основними пристроями, на які можна спиратися при написанні власних проектів.

Після відкриття вікно середовища виглядає наступним чином (рис. 2.5):

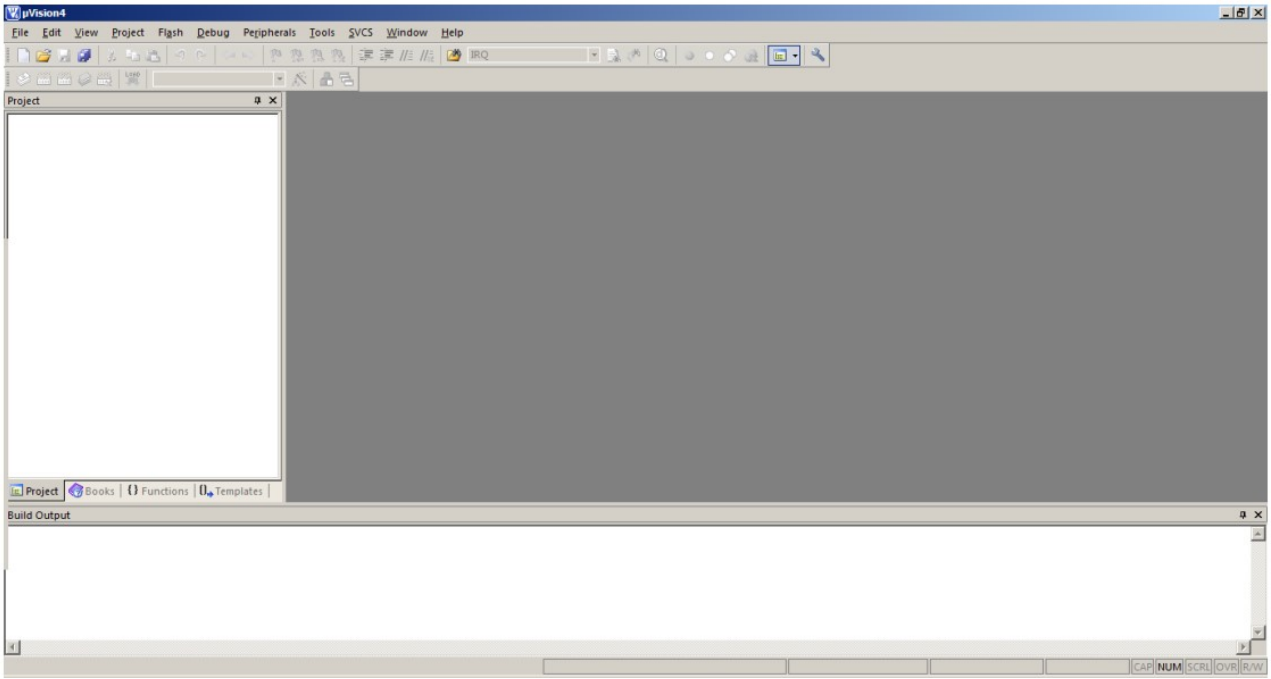


Рис. 2.5. Зовнішній вигляд середовища розробки

Тепер створимо проект для stm32F4Discovery. Для цього виконуємо команду меню Project \ New uVision Project. В результаті з'явиться діалогове вікно для вибору теки розташування проекту. Для кожного проекту бажано створювати окрему теку, оскільки проект може розростатися і містити велику кількість файлів, тому тримати 2 проекти в одній теці буде незручно. Після вибору теки слід вибрати мікроконтролер, для якого планується написання програми. Вибираємо виробника (STMicroelectronics) і конкретну модель контролера STM32F407VG (рис. 2.6).

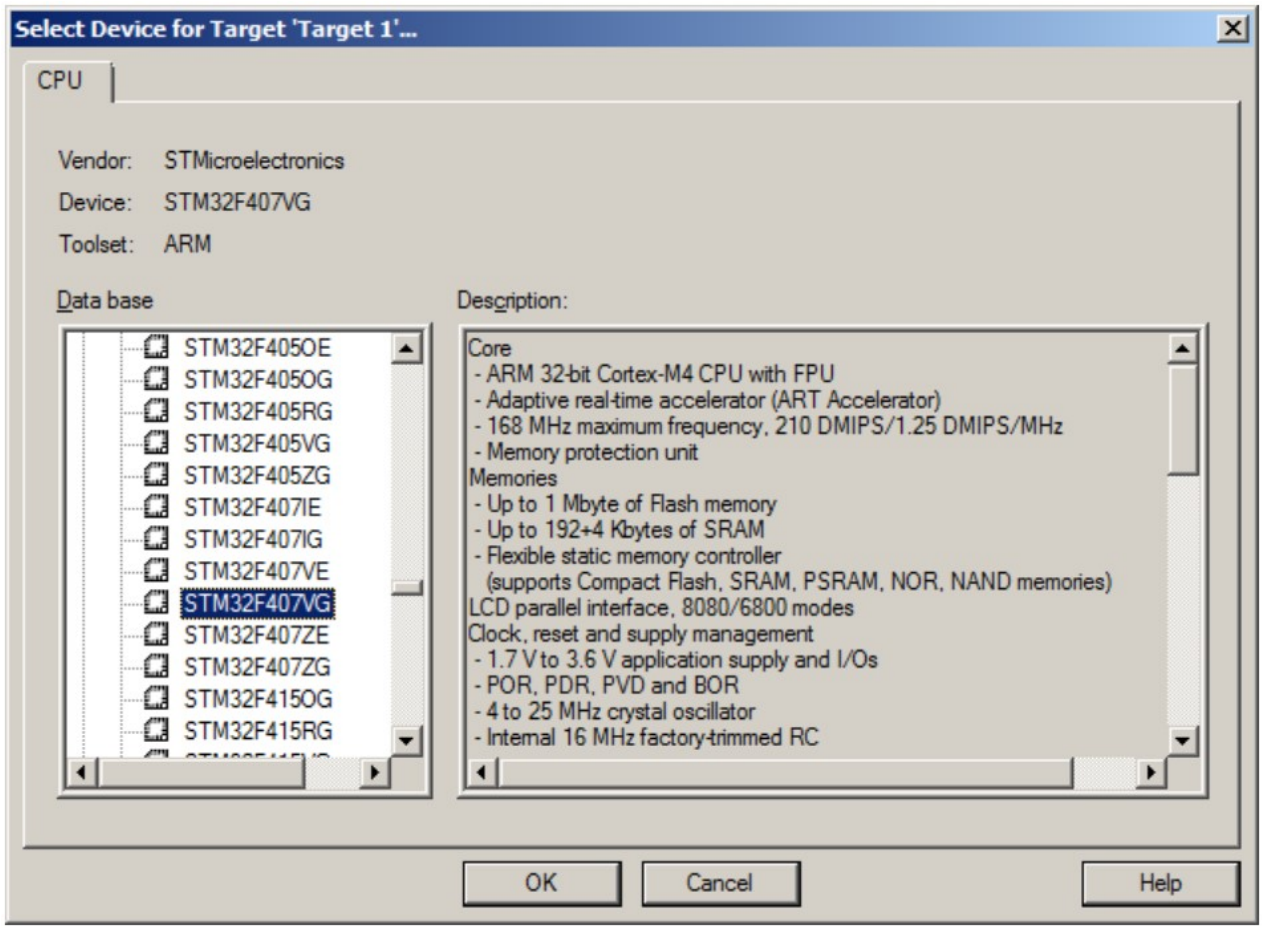


Рис. 2.6. Вибір мікроконтролеру для якого писатиметься програма

Далі відкриється діалогове вікно з пропозицією додати до проекту файл запуску startup_stm32f4xx.s, яке потрібно підтвердити. Середовище розробки створить проект, а також створить в ньому групу для вихідних файлів, в якій і буде розташований згаданий вище файл. Групи файлів служать для зручної організації представлення файлів в проекті. В даному прикладі використано 4 групи файлів: Startup, User, CMSIS і SPL.

Для початку перейменуємо створену за замовчуванням групу і назвемо її Startup та додамо ще 3 групи з допомогою контекстного меню проекту в панелі Project, виконавши команду Add Group.

Тепер додамо необхідні нам файли в проект. Перед додаванням файлів бажано скопіювати їх в директорію проекту. З бібліотеки CMSIS нам знадобляться наступні файли:

- stm32f4xx.h;

- system_stm32f4xx.h;

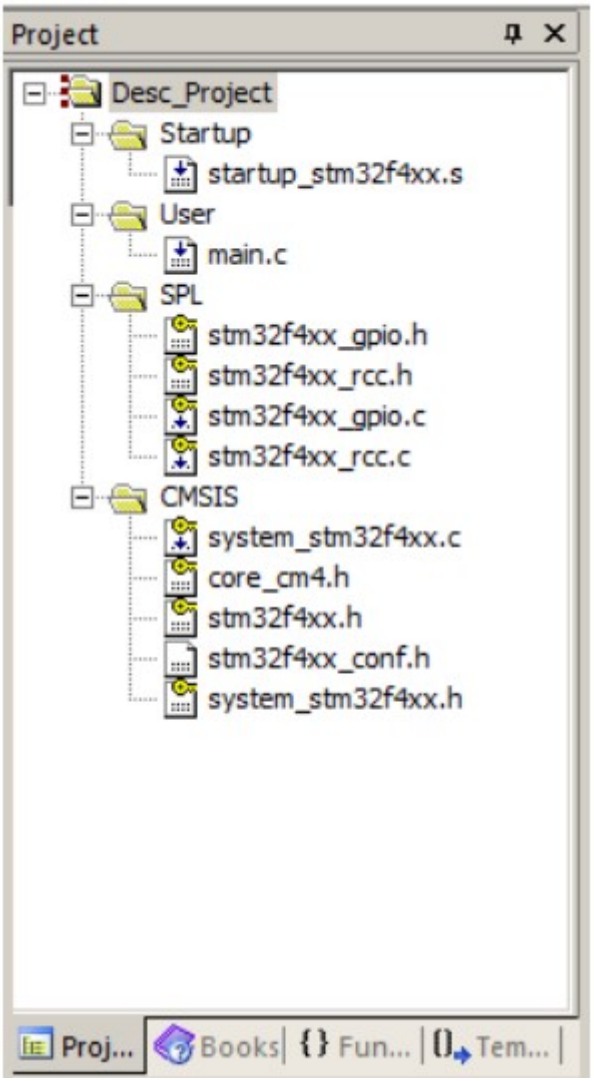
- system_stm32f4xx.c;

- stm32f4xx_conf.h;

- core_cm4.h.

Всі перераховані вище файли знаходяться в директорії, де розпакований архів з бібліотекою. До групи SPL Вам потрібно додати файли з бібліотеки SPL для роботи з пристроями. Необхідно додавати пару *.c (*.cpp) і файлів заголовків *.h (*.hpp). Наприклад, для роботи з портами введення / виводу слід додати файли stm32f4xx_gpio.c та stm32f4xx_gpio.h. Обов'язково слід підключити файли для управління пристроєм скидання і тактування, а саме, stm32f4xx_rcc.c та stm32f4xx_rcc.h. Вони містять функції для вмикання тактування периферійних пристроїв.

Після цього залишається додати ще один файл, в якому міститься функція main. Спочатку створимо і збережемо його за допомогою команд меню File, а потім додамо його в групу Users. В результаті отримаємо проект з наступною структурою (рис. 2.7):



2.7. Структура типового проекту для мікроконтролеру STM32

Залишилося зробити ще деякі налаштування, які потрібні для компіляції коду та виконання налагодження. Налаштування виробляються у вікні налаштувань, яке відкривається через команду Project / Options for target ... або кнопкою на панелі інструментів, або, використовуючи комбінацію клавішею Alt-F7. У вікні, на вкладці C/C++ задамо визначення USE_STDPERIPH_DRIVER, STM32F4XX в текстовому полі Define. Тут ж можна задати шляху до теки з заголовками файлами.

Налаштування налагодження за замовчуванням дозволяють проводити налагодження в режимі симуляції, проте, якщо пристрій підключено до комп'ютера, то краще проводити налагодження на самому пристрої. Для цього необхідно на вкладці Debug для пункту Use вказати ST-Link Debugger. Після

цього слід натиснути кнопку Settings. На вкладці Debug значення Port встановити як SW. Потім перейти до вкладки Trace Download та додати в список Programming Algorithm пристрій STM32Ff4xx як показано на рис. 2.8.

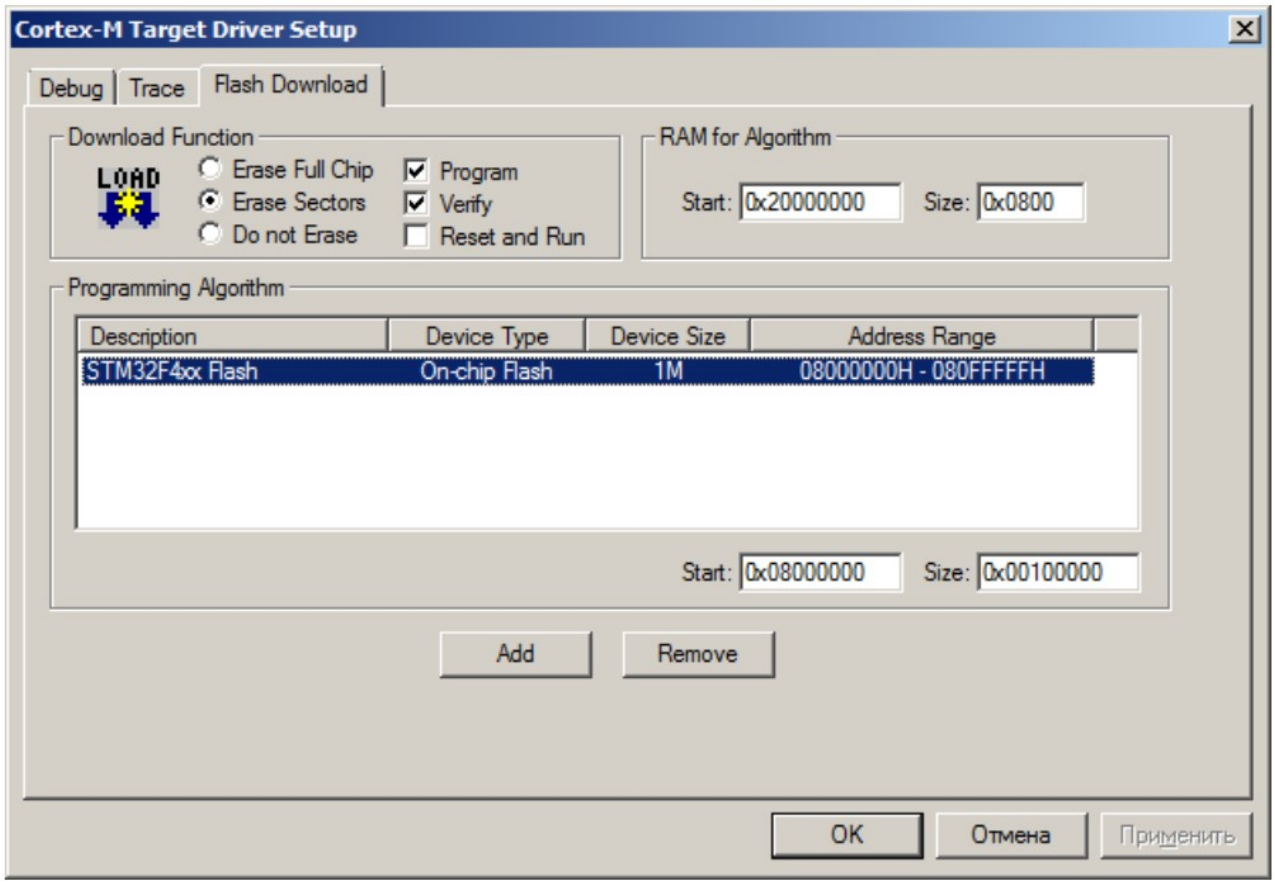


Рис. 2.8. Налаштування для налагодження програми на фізичному пристрої

На вкладці Utilities вибрати варіант Use Target Driver for Flash Programming і також встановити значення ST-Link Debugger. Перевірити, що в вікні після натискання кнопки Settings значення збігаються з заданими для налагодження.

Після проведення даних дій можна приступати до написання програми та проведення налагодження на демонстраційній платі. Однак перед цим слід підключити плату до ПК.

Для роботи з платою необхідно, щоб пройшла установка драйверів. Зазвичай, при установці засобів розробки, наприклад, Keil, відбувається також установка супутнього програмного забезпечення (якщо цього не сталося, то

слід знайти в директорії установки необхідні програми і встановити їх самостійно), тому рекомендується проводити такі дії вже після установки засобів розробки. Якщо ж варіант з установкою середовища розробки не підходить, то слід завантажити і встановити програму STM32 ST-Link Utility, яка також розглядається в методичних вказівках.

При підключенні пристрою відразу ж починається установка драйверів для нового пристрою. В області системного трею з'являється повідомлення наступного про наявність нового пристрою для якого не знайдено драйверів. Тому перед підключенням пристрою потрібно встановити USB драйвер St-Link, який можна знайти на офіційному сайті st.com: https://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-utilities/stsw-link009.html.

2.2. MicroXplorer

Виробник мікроконтролерів STMicroelectronics пропонує програму з можливістю графічної настройки мікроконтролерів серії STM32 - MicroXplorer. Вона дозволяє отримати вихідний код ініціалізації периферійних пристроїв і відповідних висновків і відображає результат настройки (зайняті виходи, режим роботи та інші). Проте, з її допомогою можна налаштувати роботу з перериваннями для конкретного пристрою, провести ініціалізацію пристроїв, що входять до складу контролера, тому даний інструмент більше підходить для наочності при розробці, а не як повноцінна альтернатива написання коду ініціалізації.

Програма представляє собою плагін до середовища розробки Eclipse. Тому попередньо необхідно встановити Eclipse, а потім додати до неї плагін.

Eclipse можна скачати з сайту http://eclipse.org. При цьому завантажується архів, який необхідно розпакувати і створити потрібні ярлики для запуску. Для

використання плагіну підходять найновіші версії програмного продукту. Сам плагін можна доступний для завантаження на сайті виробника http://www.st.com/web/en/catalog/tools/PF257931 (оскільки адреса може змінитися, то, можливо, слід скористатися пошуком по сайту). В результаті також завантажується архів для розпакування.

Установка розширення виробляється з середовища розробки. Для цього вибираємо команду меню Help \ Install New Software. У вікні, натискаємо кнопку Add і в діалоговому вікні, натиснувши кнопку Archive, вказуємо шлях до розширення. Називаємо отримане розташування і переходимо до процесу установки. Відзначаємо пункт MicroXplorer, який з'явився в списку нижче. Далі погоджуємося з умовами ліцензійної угоди і під час установки підтверджуємо запити, які з'являються від діалогових вікон. Після завершення процесу серед запропонує перезавантажити середу розробки для того, щоб застосувати зміни і завантажити розширення.

Після перезавантаження в головному меню можна побачити новий пункт - MicroXplorer (рис. 2.9):

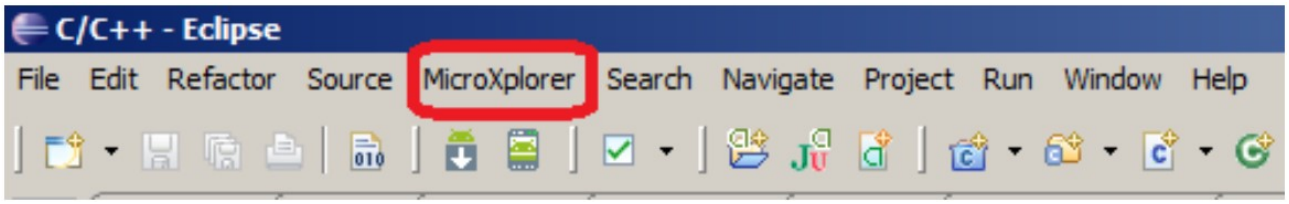


Рис. 2.9. Доданий пункт меню

За допомогою доданого плагіїну можна на вибір відкрити розширення окремого представлення Eclipse або встановити функціональність MicroXplorer доступною в усьому розташуванні (perspective) Eclipse.

Розглянемо роботу в режимі окремого подання. Виконаємо команду MicroXplorer \ Open MicroXplorer As New View. В результаті відкриється ще одне вікно, яке можна розгорнути на всю ширину робочої області двічі клацнувши по заголовку. В результаті вікно плагіну виглядає наступним чином (рис. 2.10)

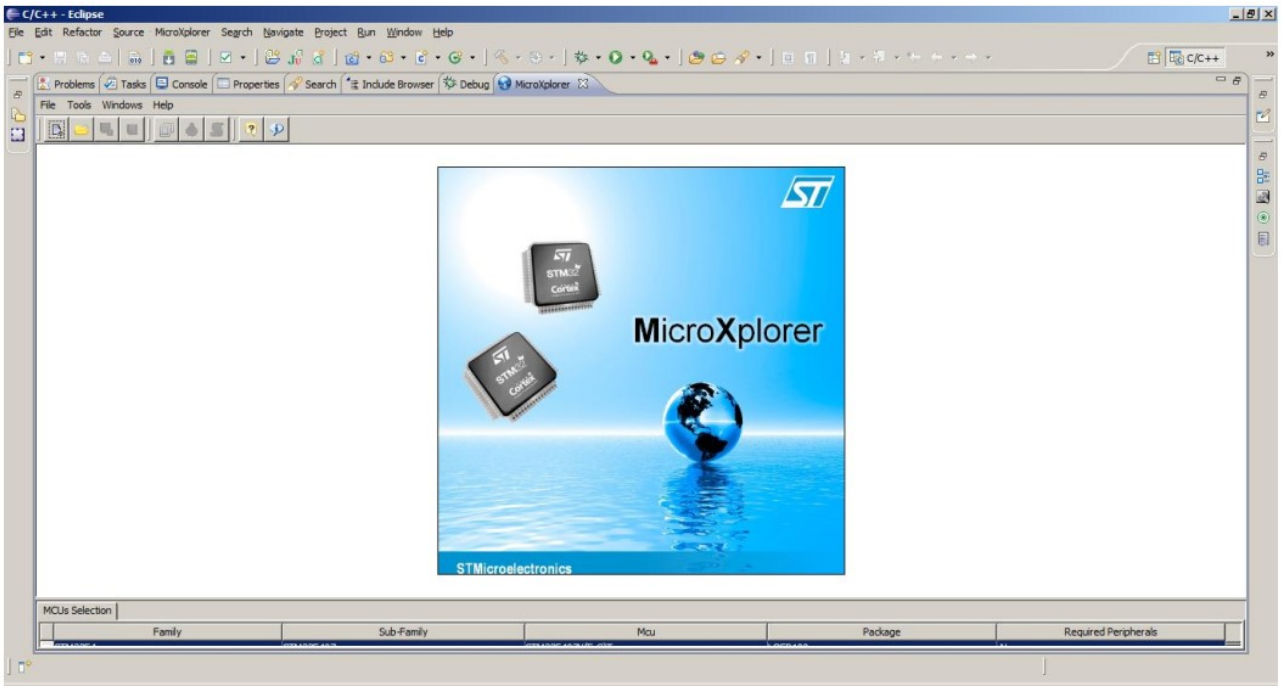


Рис. 2.10. MicroXplorer в середовищі Eclipse

Для початку проектування необхідно визначити, який саме пристрій використовується, виконавши команду Tools \ MCUs Selector. Після цього відкриється діалогове вікно (рис. 2.11), в якому зі списку вказуємо конкретний мікроконтролер і натискаємо кнопку ОК.

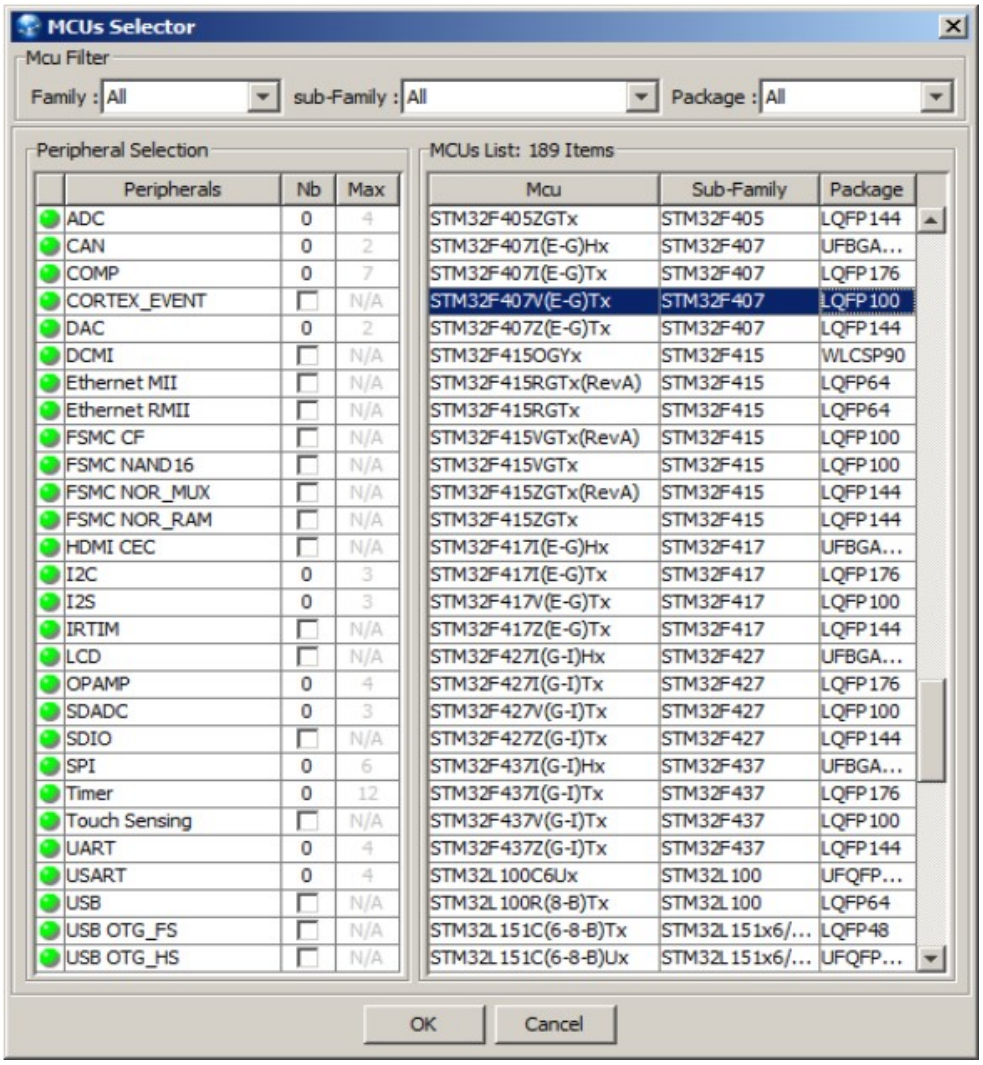


Рис. 2.11. Вікно обирання процесору

Після цього в робочій області з'явиться панель з пристроями, що входять до складу контролера, а також його умовне зображення, на якому позначені всі основні висновки (рис. 2.12). сама робоча область представлена трьома вкладками: Pinout, Configuration, Power Consumption Calculator. У міру того як відбувається налаштування і включення відповідних пристроїв, виходи з мікросхеми, які будуть задіяні, виділяються зеленим кольором. У той же час, деякі з пристроїв в панелі позначаються жовтими або червоними іконками. Це відбувається через те, що пристрої використовують виходи у своїй роботі для кількох пристроїв одночасно, тому може відбуватися частковий (використання пристрою все ще можливо) або повний конфлікт (використати пристрій не вдасться на позначених червоним кольором функціях), про що і повідомляють

відповідні іконки. Таким чином, можна виявити і уникнути подібних конфліктів уже на стадії проектування, що значно спрощує подальшу розробку.

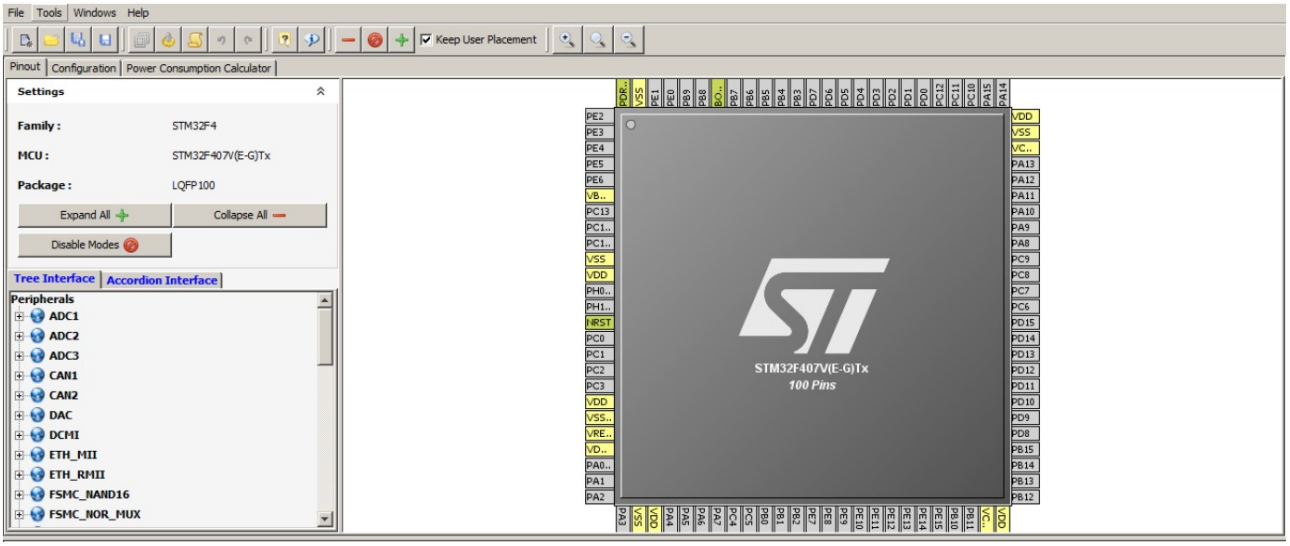


Рис. 2.12. Налаштування пристроїв процесора відповідно позначеним виходам

Розглянемо приклад налаштування першого АЦП (ADC1) в візуальному редакторі. Припустимо, що необхідно приймати сигнал з нульового входу. Цю настройку задаємо, розгорнувши вузол ADC1 і встановивши поле вибору IN0 (рис. 2.13). Висновок PA0 тепер повинен бути виділений зеленим, а біля нього з'явиться підпис ADC1_IN0. Крім того, перемістившись в панелі приладів до інших АЦП (ADC2 і ADC3), можна побачити, що вони позначені жовтими іконками. Це пов'язано з тим, що нульовий вхід всіх АЦП однаковий, тому одночасно кількома пристроями він використовуватися не може. Проте, інші АЦП все ще можуть використовуватися, але вимірювання проводити не з нульового входу.

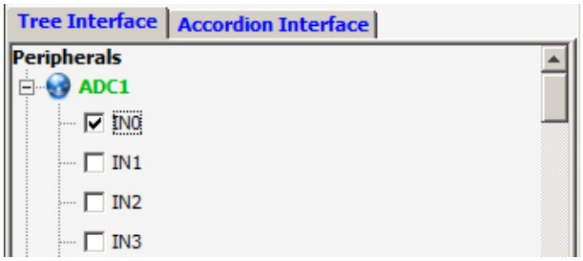


Рис. 2.13. Задання нульового каналу першого АЦП

Після цього перейдемо на вкладку Configuration (рис. 2.14). Дана вкладка призначена для налаштувань портів введення / виводу відповідно до обраної периферією контролера.

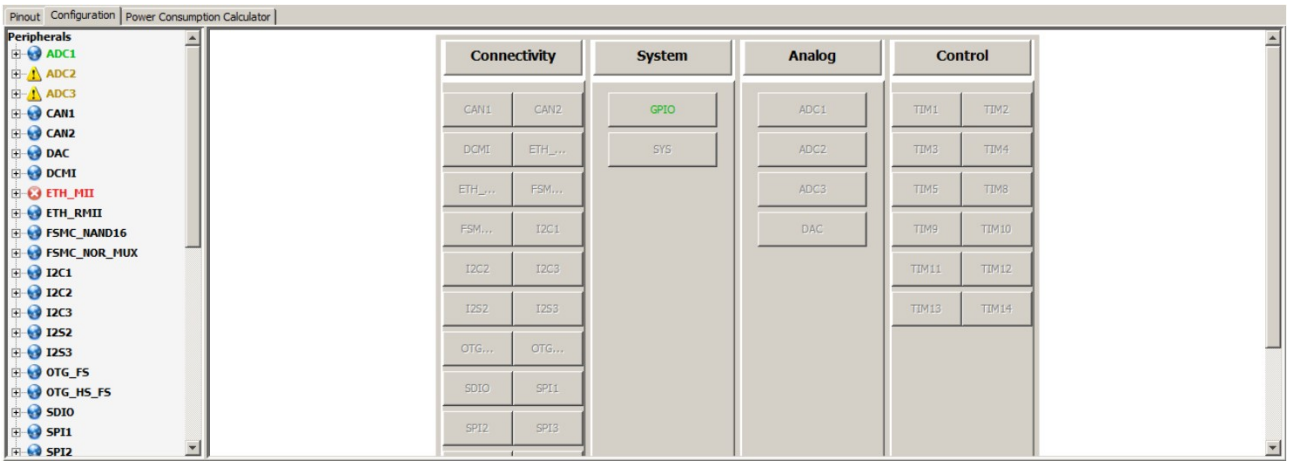


Рис. 2.14. Вкладка конфігурації контролера

Після натискання на кнопку GPIO, відкриється вікно Pin Configuration (рис. 2.15). Саме в ньому і проводиться налаштування роботи виходів мікросхеми.

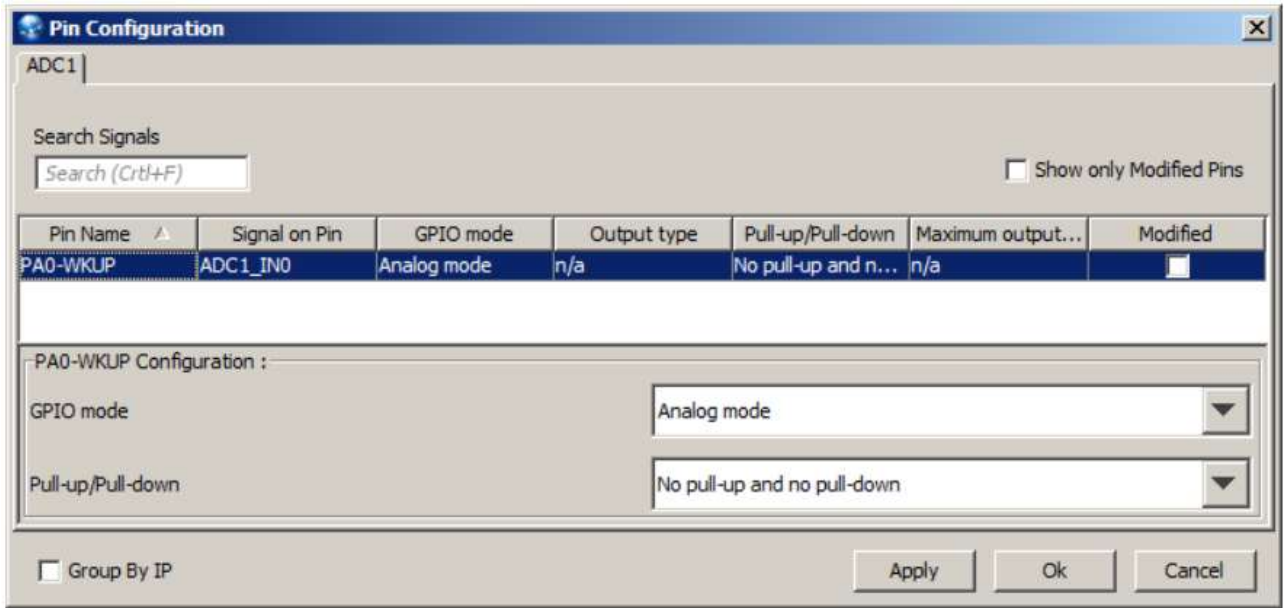


Рис. 2.15. Вікно налаштування виходів мікроконтролера

У нашому випадку згенерувала установка не вимагає змін, однак, при використанні інших пристроїв, параметри за замовчуванням слід змінювати для їх коректної роботи. При цьому на кожен обраний пристрій з'являється своя вкладка у вікні, що дозволяє відразу ж провести повне налаштування.

Завершальним етапом роботи з доповненням є генерація програмного коду ініціалізації. Вона проводиться командою меню Tools\Generate Code.... В результаті створюються теки з файлами програмного коду і файлів заголовків. Як говорилося раніше, код, що генерується програмою потрібно допрацьовувати, так як багато важливих налаштувань не генеруються автоматично. Найкраще скопіювати код, який виконує ініціалізацію портів в свою програму, а решту параметрів налаштовувати самостійно.

2.3. Налаштування тактування процесору

За замовчуванням робоча частота мікроконтролера становить 16 МГц. Цю частоту забезпечує внутрішній високочастотний генератор тактових імпульсів (High Speed Internal Oscillator - HSI). Однак максимальна робоча частота становить 168 МГц, що значно більше значення за замовчуванням. Для того, щоб налаштувати роботу пристрою на потрібній частоті виробник пропонує скористатися спеціально розробленим програмним забезпеченням - Clock Configuration Tool. Воно являє собою xls-файл з макросами, тому для його використання слід задати дозвіл виконання макросів в Excel. Інтерфейс програми (рис. 2.16) організований на одному аркуші з використанням полів введення і вибору значень і наочно відображає схему тактування, яка використовується в пристрої.

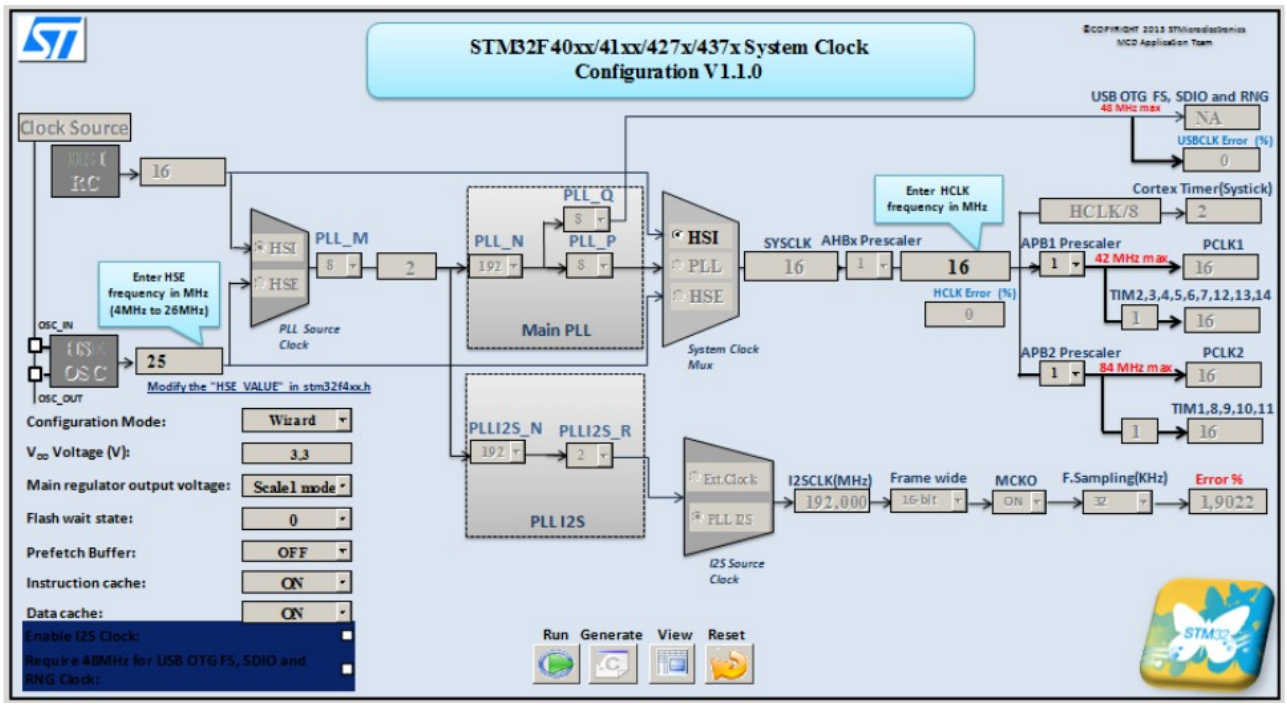


Рис. 2.16. Вигляд електронної таблиці для генерування налаштування тактування мікроконтролеру та його складових

У нижній частині схеми знаходиться кнопки, за допомогою яких запускаються макроси, які виконують основну роботу. При натисканні на кнопку Generate в теці з файлом генерується новий файл конфігурації з ім'ям system_stm32f4xx.c. Його можна включити до проекту в середовищі розробки, замінивши старий файл.

Для того, щоб застосувати налаштування слід відстежити, чи викликається функція SystemInit (), оскільки без її виклику, процесор буде працювати на частоті за замовчуванням. У разі відсутності виклику функції її слід викликати самостійно, наприклад, в основної функції main. Після цього ваш пристрій буде працювати на тій частоті, яка задана, тому, можливо, слід змінити значення параметрів роботи пристроїв для підтримки коректної роботи на обраній частоті в подальшому.

2.4. Середовище розробки CoIDE

В якості альтернативи для початківців опишемо також середовище розробки CoIDE, яка набагато простіше у використанні в порівнянні з іншими програмними продуктами, тому в її сторону слід звернути увагу абсолютним новачкам.

Для розробки програмного забезпечення під 32-розрядні процесори ARM (в тому числі і для розглянутої плати) рекомендуються такі програмні продукти:

- Atollic TrueSTUDIO;
- IAR Embedded Workbench;
- Keil µVision з інструментами MDK-ARM;
- Altium TASKING VX-Toolset.

Основною відмінною рисою всіх перерахованих середовищ є те, що вони є комерційними. Безкоштовне їх використання можливе лише з обмеженнями за розміром бінарного коду програми або за термінами безкоштовного використання.

Найбільш відомі безкоштовні альтернативи - CooCox IDE і використання Eclipse разом з плагіном для розробки для 32-розрядних процесорів. Другий варіант вимагає велику кількість додаткових налаштувань які є досить складними для тих, хто тільки починає розробку під МК або з середовищем Eclipse.

Середовище розробки CooCox CoIDE 1.7 - безкоштовний інструмент, який орієнтований на розробку для 32-розрядних процесорів ARM різних виробників: Atmel, Energy Micro, Freescale, Holtek, NXP, Nuvoton, TI. Вона побудована на основі Eclipse, однак не має таких же широких можливостей для встановлень розширень. Незважаючи на це, CooCox вимагає мінімальних налаштувань для початку програмування і налагодження, тому представляється ідеальним варіантом для початку роботи з платою STM32F4 Discovery.

Для початку розробки, крім самої CoIDE, буде потрібно установка засобів для побудови проекту GNU Toolchain for ARM Embedded Processors.

Розробка з використанням CoIDE пов'язана з концепцією сховища: всі доступні компоненти і бібліотеки встановлюються за допомогою майстра, виходячи з того, для якого саме процесора створений проект.

На жаль, сьогодні проект вже офіційно не підтримується, тому потрібно переходити на альтернативні середовища розробки та налагодження програмного забезпечення для мікроконтролерів.

Робота з проектом в середовищі CoIDE розглянута більш детально в прикладах розробки програмного забезпечення для мікроконтролерів. Вибір обумовлено вибором за найменшим періодом входження до системи розробки.

2.5. Програмне забезпечення для запису програми на мікроконтролер

Незважаючи на те, що середовища розробки інтегрують в собі функціональність по прошивці мікроконтролерів, корисно буде дізнатися, що існують і окремі програми, які призначені спеціально для роботи з пам'яттю (програмування, очищення, перевірка) пристроїв. Крім того, при роботі з режимами зниженого енергоспоживання саме ці програми дозволяють виконати програмування незалежно від поточного режиму роботи, з чим можуть виникнути проблеми при виконанні даних дій з середовища розробки.

Розглянемо два програмних продукту, які реалізують функціональність для виконання прошивки без використання середовища розробки: STM32 ST-LINK Utility та ST Visual Programmer.

Програма STM32 ST-LINK Utility призначена для роботи з 32-розрядними контролерами через інтерфейс ST-LINK (в тому числі і з його другою версією). Її інтерфейс (рис. 2.17) організований максимально просто і зрозуміло. Основна робоча область організована у вигляді двох вкладок, в яких відображається пам'ять пристрою, а також представлений двійковий файл для запису. Найбільш важливі команди зосереджені в меню Target. Якщо в момент відкриття програми пристрій не було під'єднано до ПК, то виконати підключення слід

командою Target \ Connect. Після цього в панелі повідомлень повинно з'явитися повідомлення про успішне підключенні і інформація про приєднаний пристрій. Виконати програмування пристрою можна командою Target\Program.... При цьому слід заздалегідь відкрити файл з програмою. Стирання пам'яті програм пристрою проводиться командою Target\Erase Chip

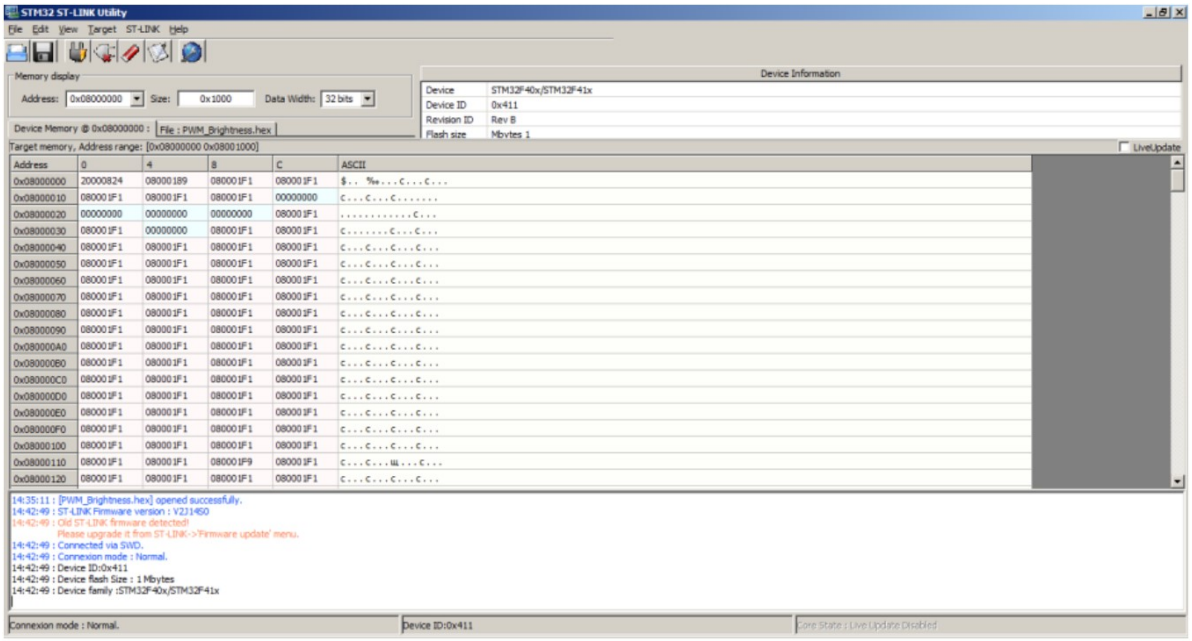


Рис. 2.17. Програма керування пам'яттю мікроконтролерів ST-LINK

Програма ST Visual Programmer (STVP) є універсальним засобом для запису програм до мікроконтролерів виробництва STMicroelectronics. Вона може працювати з пристроями сімейств STM32, STM8, а також STM7, підтримує більшу кількість інтерфейсів, що також є перевагою. В іншому ж STVP і STM32 ST-LINK Utility дуже схожі між собою як по інтерфейсу (рис. 2.18), так і по функціональності.

Використання STVP багато в чому аналогічно використанню, крім того, що необхідно самостійно задати налаштування підключення. Ці дії виконуються в діалоговому вікні (рис. 2.19), яке відкривається командою. Configure \ Configure ST Visual Programmer. У ньому вказується апаратне забезпечення, яке використовується для підключення та обирається конкретний пристрій.

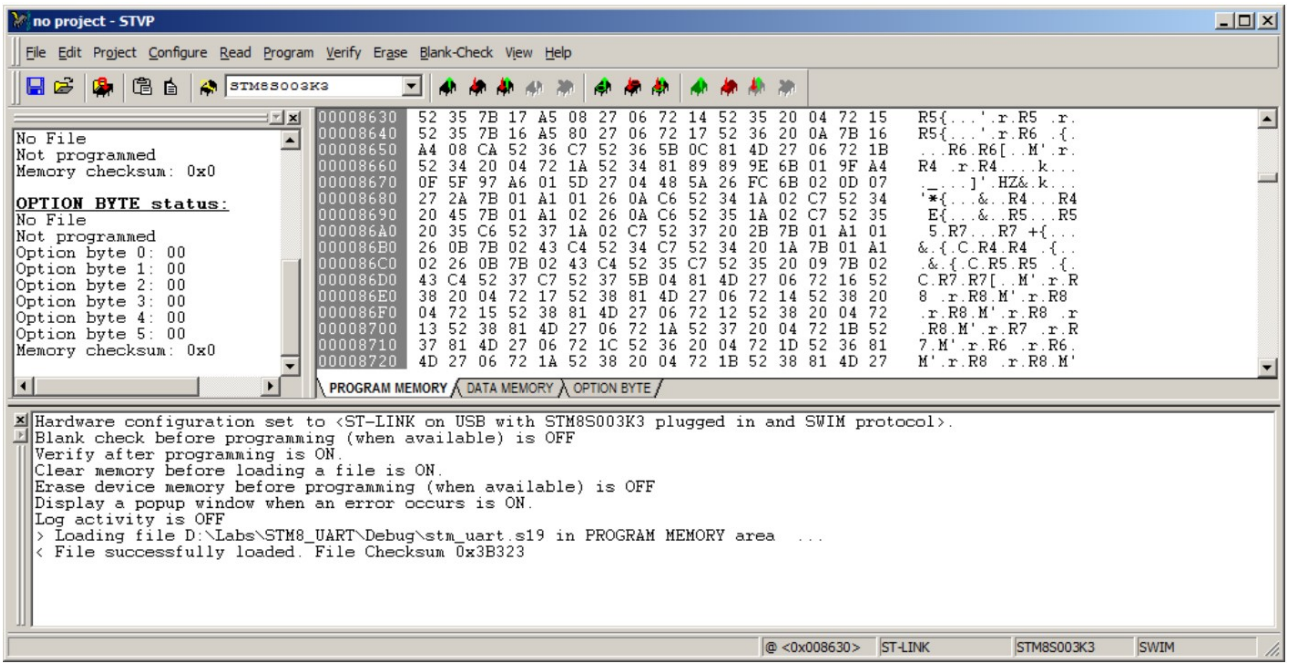


Рис. 2.18. Програма керування пам'яттю мікроконтролерів STVP

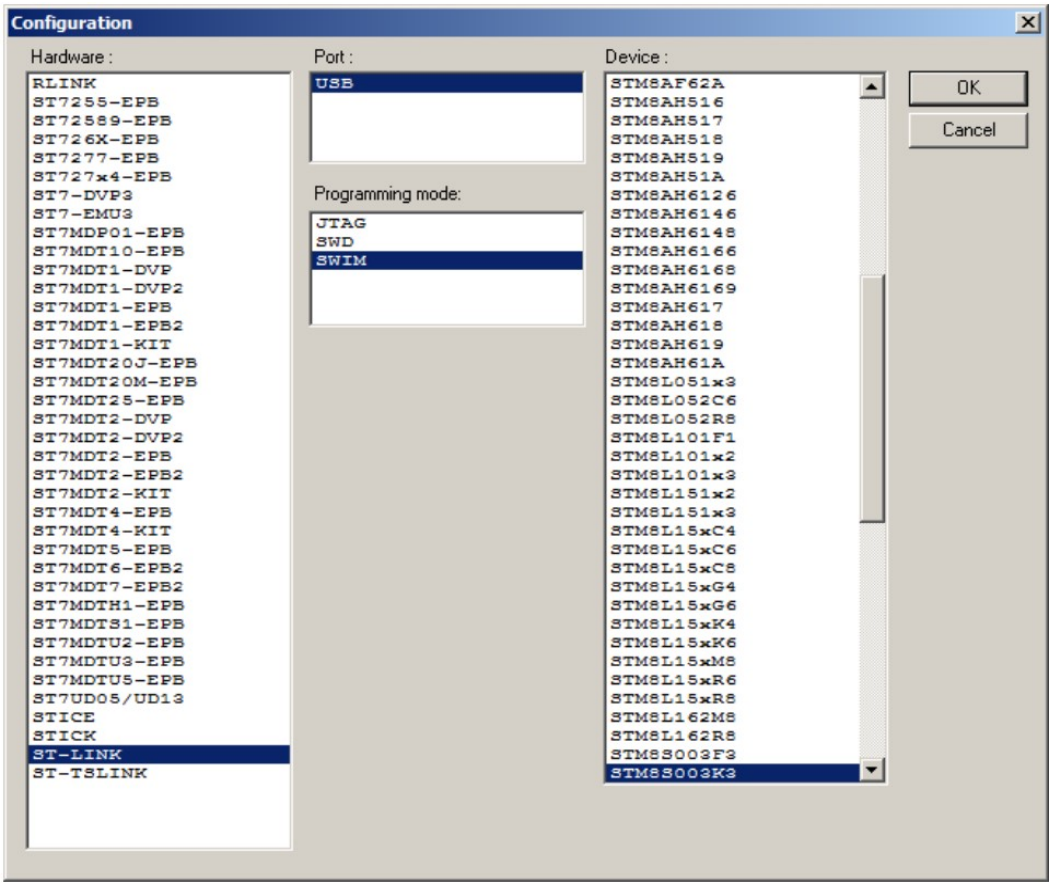


Рис. 2.19. Налаштування STVP для з'єднання з мікроконтролером

Робота налагоджування платою STM32F4 Discovery з використанням описаних інструментів відбувається за схожим сценарієм, тому перевага в їх використанні залежить від користувача. Автоматичне підключення при запуску полегшить роботу новачків, а для тих, хто працює з різними пристроями можна рекомендувати STVP.

3. ПРИКЛАДИ СТВОРЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Створення проекту в середовищі розробки. Використання портів введення / виводу

Контролер STM32F407VG містить п'ять 16-розрядних портів вводу/виводу загального призначення, які позначені як GPIOx, де x може мати значення A, B, C, D, E. Кожен порт GPIO має чотири 32-розрядних регістри конфігурації (GPIOx_MODER, GPIOx_TYPER, GPIOx_SPEEDR, GPIOx_ORD), два 32- розрядних регістри даних (GPIOx_ODR, GPIOx_IDR) і два 32-бітових регістра вибору додаткових функцій (GPIOx_AFRH і GPIOx_AFRL).

Світлодіоди, призначені для програмування, на платі підключені до порту D (рис. 3.1).

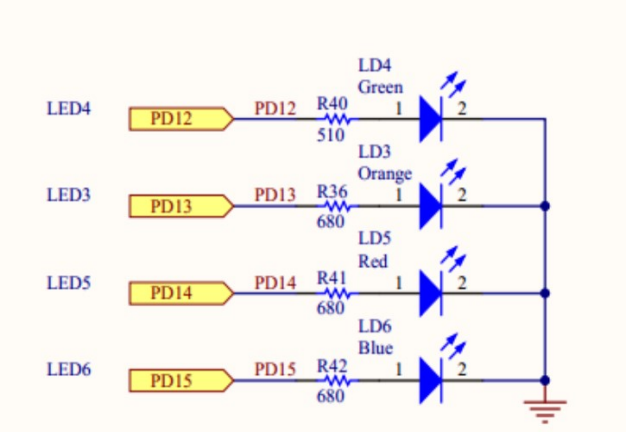


Рис. 3.1. Фрагмент принципової електричної схеми платі налагодження

Зі схеми видно, що для світіння світлодіодів потрібно через виходи мікроконтролера PD12-PD15 подавати напругу +, що відповідає логічному стану “1”. При поданні на вихід логічного стану “0” світіння спостерігатися не буде.

Решта чотири світлодіода виконують службові функції індикації та в програмуванні певних дій не використовуються.

Початківцям рекомендується встановити середовище розробки CooCox CoIDE 1.7. Створення проекту виконується за допомогою майстра, який відкривається при виконанні команди меню Project / New Project. Далі слід покроково вказати ім'я проекту і його розташування, виробника і МК, для якого призначена програма. Далі робота з проектом здійснюється за допомогою вікна Repository, яка дозволяє додати необхідні бібліотеки управління периферійними частинами МК, а також вікно Project. Вікно Repository також є майстер, який містить кілька вкладок, основний з яких є вкладка Peripherals (рис. 3.2).

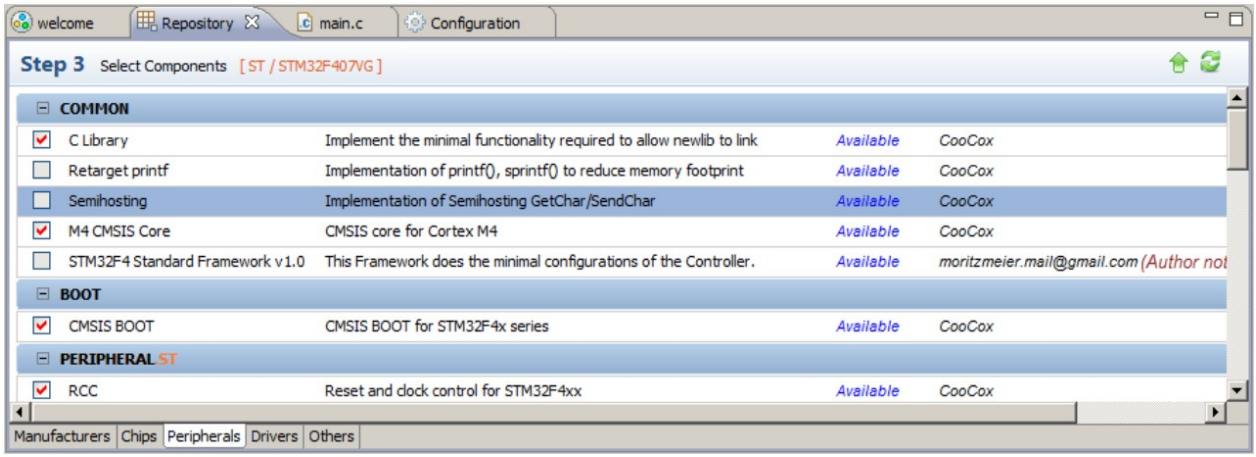


Рис. 3.2. Додавання модулів до проекту

Проект програми для плати STM32F4 Discovery має схожу структуру для всіх засобів розробки. Зазвичай він включає в себе два внутрішніх каталоги: один для файлів з бібліотеки CMSIS, а інший для файлів роботи з периферією. Для прикладу представлені типові структури проектів в середовищах розробки CoIDE і Keil (рис. 3.3).

Розглянемо приклад програми, яка демонструє роботу з портами введення/виводу. вона дозволяє перемикати стану світлодіода при натисканні кнопки, яка знаходиться на платі. Кнопка замикає через додатковий опір вихід A0 на джерело живлення, тим самим подає високий логічний рівень в момент утримання кнопки.

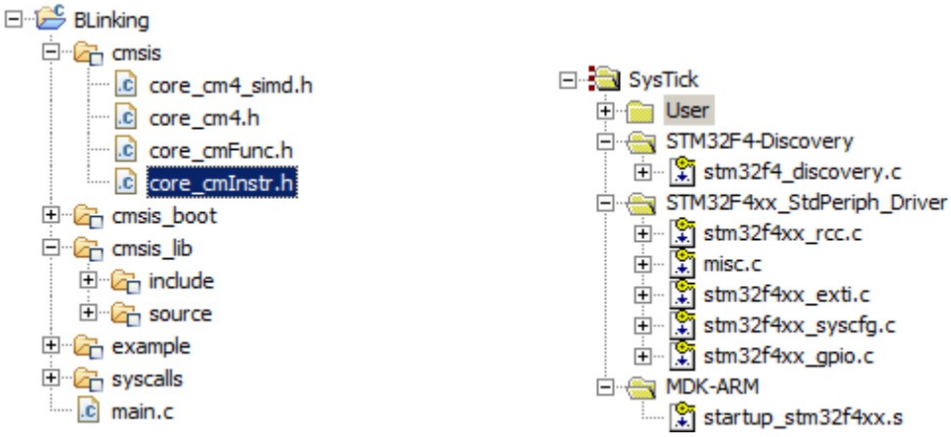


Рис. 3.3. Дерево проектів в CoIDE та Keil

Розглянемо детально наступний проект.

На початку файлу main.c оголошуємо використання наступних бібліотек:

```
#include "stm32f4xx.h"
```

Бібліотека містить оголошення регістрів конкретного процесору.

```
#include "stm32f4xx_rcc.h"
```

Бібліотека містить структури та функції керування тактуванням периферійних пристроїв мікроконтролеру.

```
#include "stm32f4xx_gpio.h"
```

Наступна бібліотека містить керування виходами мікросхеми.

```
const int LED1 = GPIO_Pin_12;
const int LED2 = GPIO_Pin_13;
```

Для зручності позначимо виходів мікросхеми, які з'єднано до світлодіодів, константами. Це дає переваги при зміні процесору або пристрою, тоді потрібно змінити в програмі значення констант, і програма знову придатна для використання на новому пристрої.

Далі по коду потрібно реалізувати функції, які буде використовувати програма. Реалізуємо організацію пауз для уповільнення роботи окремих ділянок програмного коду, для цього використаємо пустий цикл. Цей цикл змушує процесор працювати “в холосту”, при цьому фактична пауза може значно різнитися від налаштувань процесора. Слово **volatile** використано для позначення компілятору, що спрощувати вирази з цією змінною не можна, її значення може змінитися непередбачувано, наприклад, у перериванні. Завдяки цьому при увімкненому оптимізаторі коду, компілятор не замінить цей цикл на просте присвоєння нуля змінній.

```
void Delay(volatile unsigned int nCount)
{
    while(nCount--);
}
```

Наступна функція призначена для налаштування виходу з мікросхеми PA0 як приймач інформації. Для цього використана структура GPIO_InitTypeDef, яка містить опис зрозумілою мовою всіх властивостей виходів мікросхеми (але керування альтернативними функціями винесено окремо). Далі за допомогою RCC команди подається тактувальний сигнал до периферійного пристрою GPIOA, після чого його можна використовувати. Розглянемо послідовно властивості виходу мікросхеми.

gpioConf.GPIO_Pin = GPIO_Pin_0 — вказується що тут використовується лише один вихід мікросхеми, а саме за номером нуль.

gpioConf.GPIO_Speed = *GPIO_Speed_100MHz* — швидкість спрацювання зміни логічного рівня, чим швидше працює вихід, тим більше він споживатиме енергії.

gpioConf.GPIO_PuPd = *GPIO_PuPd_DOWN* — керує значенням яке читається на вході по умовчанню, коли вихід не приєднано до джерела сигналу. Коли кнопка не натиснута, потрібна гарантія надходження логічного нуля, тому вихід “придушено” на “землю”.

gpioConf.GPIO_OType = *GPIO_OType_PP* — визначає метод керування

напругою на виході.

gpioConf.GPIO_Mode = *GPIO_Mode_IN* — задає направлення руху інформації, з кнопки до мікросхеми надходить інформація — лапка є входом інформації.

На останнє, проводиться застосування вказаних налаштувань:

```
void ButtonConfig(){
    GPIO_InitTypeDef gpioConf;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
    gpioConf.GPIO_Pin = GPIO_Pin_0;
    gpioConf.GPIO_Speed = GPIO_Speed_100MHz;
    gpioConf.GPIO_PuPd = GPIO_PuPd_DOWN;
    gpioConf.GPIO_OType = GPIO_OType_PP;
    gpioConf.GPIO_Mode = GPIO_Mode_IN;
    GPIO_Init(GPIOA, &gpioConf);
}
```

Наступна функція відповідає на налаштування виходів мікросхеми на керований вивід напруги -керування світлодіодами. Тіло функції є аналогічним, але тут вказується одночасно два виходи, які поєднуються операцією логічного бітового додавання. Також відмінним є вимкнена підтяжка виходів, бо невизначеності читання стану тут немає.

```
void LedConfig(){
    GPIO_InitTypeDef gpioConf;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
    gpioConf.GPIO_Pin = LED1 | LED2;
    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);
}
```

Наступні більшість функцій є короткими та мають призначення в заміні загального коду на змістовні назви. Компілятор визначає складність функцій, і коли це є вигідним, замість виклику функцій буде вставляти сам код. Тому уповільнення роботи програми за черезмірного розбиття на функції немає. В цьому коді проходить звернення до PA0 з визначенням напруги. Якщо на вхід подається напруга, функція вертатиме 1, інакше — 0.

```
int GetButtonState(){
    return GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0);
}
```

Наступна функція не може завершити цикл доки стан кнопки повертає 1, тобто, вийти з цього циклу програма зможе лише при відпусканні кнопки:

```
void WaiteButtonRelease(){
    while(GetButtonState()!=0);
}
```

Далі функція визначає повний цикл натискання кнопки, натиснення та відпускання. Лише після цих двох подій функція передасть виконання на основну програму:

```
void WateClickButton(){
    while(GetButtonState()==0);
    Delay(10000);
    WaiteButtonRelease();
}
```

Використання команд встановлення напруги на виході є простим, але для світлодіодів, матриць та двигунів така команда виглядає однаково. Тому оформимо команду увімкнення світлодіоду:

```
void LedOn(int LED){
    GPIO_SetBits(GPIOD, LED);
}
```

Та вимкнення світлодіоду:

```
void LedOff(int LED){
```

```
    GPIO_ResetBits(GPIOD, LED);
}
```

Додатковим бонусом до такого стилю запису команд керування світлодіодами є те, що в деяких схемах світлодіоди світяться при поданні на вихід низького рівня. В такому випадку в програмному коді потрібно переписати лише дві функції, а в противному випадку — код потрібно змінювати в усіх випадках керування світлодіодами по тексту програми.

Розглянемо головну функцію програми:

```
int main(void)
{
    SystemInit();
    SystemCoreClockUpdate();
    ButtonConfig();
    LedConfig();
    while(1) {
        if(GetButtonState()==0){
            LedOff(LED1);
        }else{
            LedOn(LED1);
        }
    }
}
```

Завдяки описаним вище функціям, програма читається легко. Розглянемо лише дві перші функції. SystemInit() - відповідає за налаштування тактування процесору. Без виклику цієї функції процесор працюватиме на частоті по замовчанню — 16МГц. SystemCoreClockUpdate() - відповідає лише за розрахунок дійсної частоти процесора. Ця константа може бути використана для розрахунку частот інтерфейсів та таймерів, щоб програма рахувала час при різних налаштуваннях правильно.

В основному циклі програми відбувається зчитування стану кнопки, і

якщо кнопка натиснена, вмикається світлодіод.

Наступний головний цикл перемикає світлодіоди по натисканні кнопки:

```
while(1) {
     LedOn(LED1);
     LedOff(LED2);
     WateClickButton();
     LedOff(LED1);
     LedOn(LED2);
     WateClickButton();
}
```

На основі отриманих даних виконайте самостійне завдання, коли при натиснканні користувацької кнопки, програма буде перемикати світлодіод. Задійте всі чотири світлодіоди.

3.2. Переривання та використання таймерів

Переривання — механізм, який дозволяє апаратному забезпеченню повідомляти про настання подій у своїй роботі. У момент, коли відбувається переривання, процесор перемикається з виконання основної програми на виконання відповідного обробника переривань. Як тільки виконання обробника завершено, триває виконання основної програми з місця, в якому вона була перервана.

Для використання переривань необхідно спочатку налаштувати регістр, який називається Nested Vector Interrupt Controller (NVIC), вбудований в контролер контролер переривань. Цей регістр є стандартною частиною архітектури ARM і зустрічається на всіх процесорах, незалежно від виробника. NVIC розроблений таким чином, що затримка переривання мінімальна. NVIC підтримує вбудоване переривання з 16-ма рівнями пріоритету.

Мікроконтролер STM32F407VG містить 14 таймерів. У загальному

вигляді схема управління рахування імпульсів може бути представлена наступним чином (рис. 3.4):

Виробник розділяє всі таймери на три типи:

1) з розширеними можливостями;

2) загального призначення;

3) базові.

Кожен таймер може мати до 4 ліній захоплення / порівняння (саме вони використовуються в режимі генерації ШІМ).

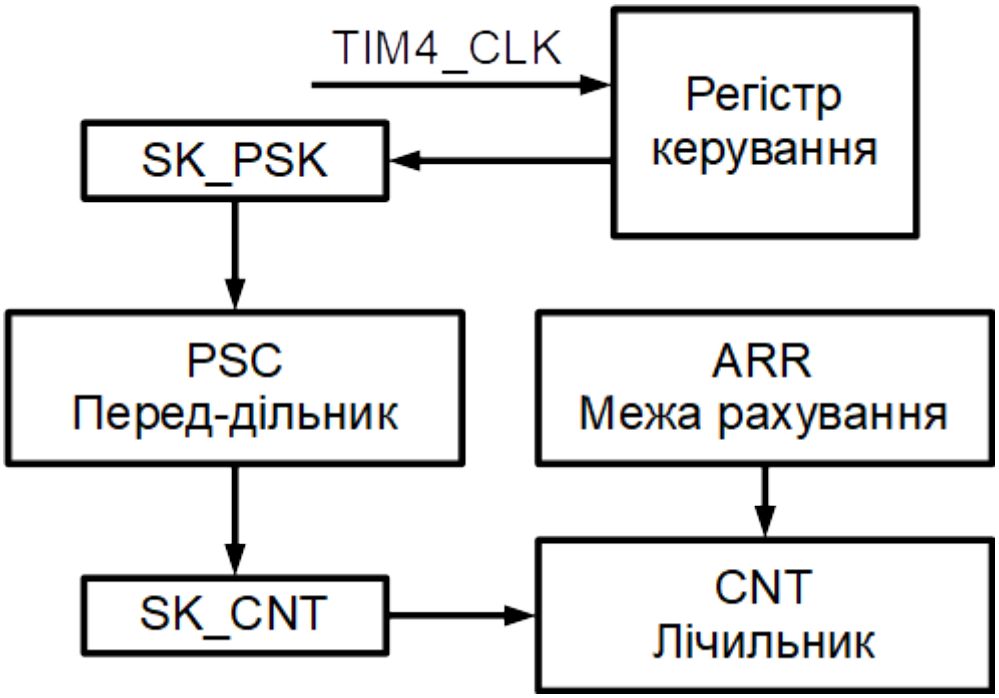


Рис. 3.4. Схема керування тактуванням таймеру

Наступна програма демонструє роботу з перериваннями і з таймерами. У ній реалізовано перемикання світлодіода, який підключений до порту введення/виведення, через кожну секунду. Для цього попередню програму доповнимо функціями:

```
void LedToggle(int LEDS){
     GPIO_ToggleBits(GPIOD, LEDS);
}
```

Функція виконує перемикання стану виходу на протилежне значення, якщо на вихід подавалася напруга то вона вимкнеться, якщо ж напруги не було

— напруга буде подана.

```
void TIM2_IRQHandler(void) {
	if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET) {
		TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
		LedToggle(LED2);
	}
}
```

Наведена функція має фіксовану назву обробника переривання. Всі доступні обробники переривань можна переглянути у файлі startup секції. Для того щоб визначити власний обробник переривання, достатньо створити функцію з відповідною назвою. Також для багатьох пристроїв переривання може викликатися по різним подіям, тому в обробнику передбачено умовний оператор в якому перевіряється факт переповнення лічильнику, лише після цього становиться ознака, що переривання оброблено, а потім змінюється стан другого світлодіоду.

Наступна функція складається з двох логічних частин, налаштування дозволу на переривання та налаштування таймеру:

```
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 = SystemCoreClock/10000 - 1;
	tim_struct.TIM_ClockDivision = TIM_CKD_DIV1;
	tim_struct.TIM_CounterMode = TIM_CounterMode_Up;
	TIM_TimeBaseInit(TIM2, &tim_struct);
	TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
	TIM_Cmd(TIM2, ENABLE);
}
```

Налаштування дозволу переривання відбувається викликом функції NVIC_Init, для якої підготовлені налаштування в структурі NVIC_InitTypeDef. Для по використанню в лістингу функції, вказується вектор переривання, пріоритет, підпріоритет та команда задіяння переривання.

Наступний блок відповідає за роботу таймеру. Після увімкнення тактування таймеру заповнюється структура в якій зазначено, що період лічильника є 10000-1=9999, це означає доступність значення лічильника в межах 0..9999 — десять тисяч циклів (бо нульовий стан теж потрібно враховувати). Наступним параметром налаштовується передлічильник, який фактично виступає в ролі дільника частоти. По досягнені попереднього лічильника вказаного значення, його таймер скидає на нуль та на лічильник відправляє імпульс. Тут SystemCoreClock/10000-1 гарантує, що 10000 імпульсів, які складають період, вкладатимуться в одну секунду. Це означає щосекундний виклик обробника переривань (якщо частота процесору налаштовано правильно).

Далі застосовуються налаштування, вмикається генерування події переповнення лічильника та подається команда на увімкнення таймеру. Саме з цього моменту починається рахування часу та виклик обробника переривання.

Тіло програми тепер матиме вигляд:

```
int main(void)
{
	SystemInit();
	SystemCoreClockUpdate();
	LedsInit(LED1 | LED2 | LED3 | LED4);
	LedOff(LED1 | LED2 | LED3 | LED4);
	INTTIM_Config();
	while(1){
		LedOn(LED1); LedOff(LED3);
		Delay(500000);
		LedOn(LED3); LedOff(LED1);
		Delay(500000);
	}
}
```

Ця програма мало відрізняється від попередньої, додано лише виклик налаштування обробника переривання, після якого відбувається увімкнення світлодіоду кожні дві секунди.

Для самостійного опанування, модифікуйте програму з попередньої глави, коли перемикання світлодіодів відбувається не лише за натисканням кнопки, але й по події таймеру.

3.3. Генерування сигналу широтно-імпульсної модуляці

Широтно-імпульсна модуляція (ШІМ) - спосіб управління середнім значенням напруги на навантаженні шляхом зміни ширини імпульсів. В основному, мікроконтролери дозволяють генерувати цифровий ШІМ різної частоти.

Мікроконтролер призначений для керування зовнішніми пристроями за алгоритмами, які вираховують сигнали керування за отриманими з датчиків даними. Деякі пристрої вимагають для керування формування сигналів широтної імпульсної модуляції (ШІМ). Зміст цього сигналу показано на наступному малюнку:

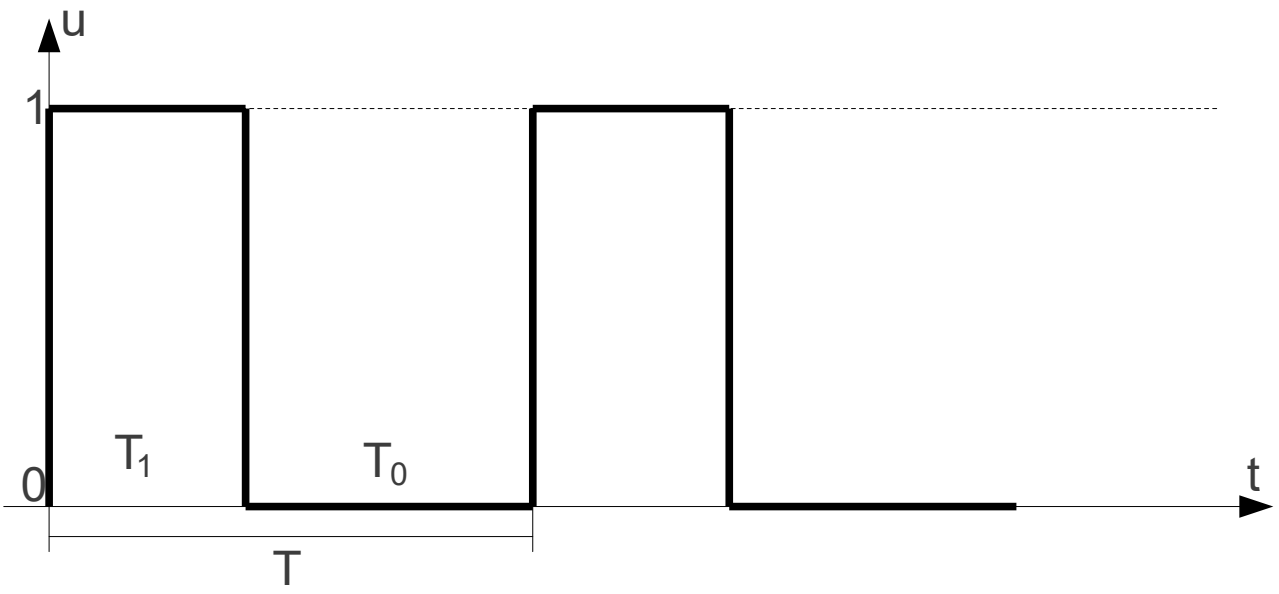


Рис. 3.5. Приклад ШІМ сигналу

ШІМ сигнал формується одиничним імпульсом напруги з певною паузою. При цьому час T=const, період сигналу є величиною постійною, а за цей час розподіл T_1, T_0 може змінюватися. Та завжди є справедливою сума $T=T_1+T_0$.

Такий сигнал може використовуватися для регулювання світимості світлодіоду або потужності двигуна постійного струму. Відомо, що світлодіод має порогову напругу живлення, після якої він починає майже одразу світитися

на повну потужність, і з метою запобігання перегорання, струм живлення та яскравість світіння обмежують додатковим опором. Для регулювання світності світлодіоду потрібно використати змінний опір, але опір мікроконтролером регулювати є нетривіальною задачею, тому світлодіод використовують в режимі максимальної потужності, але живлять його пульсуючим струмом ШІМ сигналу. Фактично за час T світлодіод на повну потужність світить час T_1, тобто $100T_1/T$ % часу світлодіод є джерелом світла. Для малого періоду T, коли за одну секунду відбувається більше 1000 спалахів, людське око сприймає такий режим як плавну зміну яскравості світлодіоду. Ще однією з переваг такої схеми живлення є більш високе ККД використання електричної енергії, зменшивши її витрачання на додаткових опорах.

Перед підключенням пристрою відомі такі параметри ШІМ, як частота і коефіцієнт заповнення. Для їх розрахунку в контролерах STM32 необхідно визначити значення преддільника і автоматично завантажувати значення в регістрі ARR (Auto-Reload Register). Розрахунок значення, яке слід записати в переддільник виконується наступним чином:

$$PSC = \frac{TIMxCLK}{TIMxCNT} - 1,$$

де *PSC* – значення переддільника;

TIMxCLR – вихідна частота роботи таймеру;

TIMxCNT – частота лічильника.

Для отримання необхідної вихідної частоти слід записати значень в регістр ARR, яке повчає з наступного співвідношення:

$$\text{A}RR\_VAL = \frac{TIMxCNT}{TIMx\_вихідна\_частота} - 1,$$

де *ARR_VAL* – значення для запису в регістр *ARR*;

TIMx_вихідна_частота – вихідна частота роботи таймеру;

TIMxCNT – частота лічильника.

Останнім етапом є завдання потрібного коефіцієнта заповнення, що забезпечить потрібне заповнення імпульсу. Ця процедура проводиться за допомогою регістра захоплення/порівняння (Capture / compare register, CCRx), виходячи з наступного співвідношення:

$$Z = \frac{CCRx\_VAL}{ARR\_VAL}100\%,$$

де Z – коефіцієнт заповнення, $CCRxVAL$ – значення регістру $CCRx$.

Особливістю даних мікроконтролерів є те, що в переддільник і інші регістри можна записати будь-яке значення, яке можна описати з допомогою відведеного кількості розрядів. Видача сигналу ШІМ на вихід не є основним режимом роботи висновків порту, а відноситься до додаткових (альтернативним) режимам. Тому попередньо потрібно задати потрібний режим в налаштуваннях порту. Для підключення альтернативних функцій до портів введення / виводу необхідно:

1. Підключити вихід до альтернативної функції відповідного периферійного пристрою за допомогою функції GPIO_PinAFConfig.

2. Конфігурувати вихід в режим виконання альтернативної функції - GPIO_Mode_AF.

3. Виконати інші налаштування.

4. Викликати GPIO_Init для застосування зазначених налаштувань.

Розглянемо програмну реалізацію:

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

Наступна функція відповідає за налаштування четвертого таймеру на генерування ШІМ по чотирьом каналам:

```
void PWM_Init(){
     TIM_TimeBaseInitTypeDef time_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 = TIM_CKD_DIV1;
    time_init.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM4, &time_init);

    TIM_OCInitTypeDef oc_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);
    TIM_OC2Init(TIM4, &oc_init);
    TIM_OC2PreloadConfig(TIM4, TIM_OCPreload_Enable);
    TIM_OC3Init(TIM4, &oc_init);
    TIM_OC3PreloadConfig(TIM4, TIM_OCPreload_Enable);
    TIM_OC4Init(TIM4, &oc_init);
    TIM_OC4PreloadConfig(TIM4, TIM_OCPreload_Enable);
    TIM_ARRPreloadConfig(TIM4, ENABLE);
    TIM_Cmd(TIM4, ENABLE);
}
```

В першу чергу командою RCC_APB1PeriphClockCmd подається тактування на четвертий таймер. Далі розраховується значення передлічильника PrescalerValue так, щоб частота ШІМ імпульсів не залежала від тактової частоти процесора. Далі встановлюємо періодичність імпульсів в 665 тактових імпульси з передлічильника. Інші налаштування є аналогічними що до налаштування таймеру 2, який розглянуто в попередньому прикладі.

Далі в функції PWM_Init проводиться налаштування виходів ШІМ, окремо для кожного з каналів. Для цього проводиться заповнення структури TIM_OCInitTypeDef. IM_OCMode приймає значення TIM_OCMode_PWM1, бо саме цей режим відповідає за генерування неперервних імпульсів заданої ширини. TIM_OutputState приймає значення TIM_OutputState_Enable для увімкнення подання сигналу на вихід мікроконтролера. TIM_Pulse = 0 визначає початкову ширину сигналу, а TIM_OCPolarity = TIM_OCPolarity_High визначає, що регулювання відбувається шириною саме початкового одиничного імпульсу, а до кінця періоду буде виводитися нульовий рівень. Після цього застосовуються налаштування на кожен з каналів. Завершує функцію вмикання четвертого таймеру, після чого починається генерування сигналу.

Налаштування виходів до світлодіодів відрізняється від простого

використання GPIO:

```
void LedsPWMInit(){
	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_15;
	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);
	GPIO_PinAFConfig(GPIOD, GPIO_PinSource13, GPIO_AF_TIM4);
	GPIO_PinAFConfig(GPIOD, GPIO_PinSource14, GPIO_AF_TIM4);
	GPIO_PinAFConfig(GPIOD, GPIO_PinSource15, GPIO_AF_TIM4);
}
```

Тут режим роботи обрано як GPIO_Mode_AF, що означає використання виходу в одному з альтернативних режимів. Після застосування налаштувань виходів до світлодіодів, вказується для кожного з виходів, яку саме альтернативну функцію вони будуть використовувати.

Наступна змінна показую фазу розподілу яскравості між світлодіодами:

```
volatile int faze = 0;
```

Наступна функція є обробником переривання від другого таймеру, вона використана для зміни фази світіння світлодіодів та запису ширини імпульсів у регістри CCRx таймеру, завдяки цьому ширина імпульсів по чотирьом каналам є різною:

```
void TIM2_IRQHandler(void) {
	if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET) {
		TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
		faze = (faze+1)%665;
		TIM4->CCR1 = (faze+665*0/4)%665;
		TIM4->CCR2 = (faze+665*1/4)%665;
		TIM4->CCR3 = (faze+665*2/4)%665;
		TIM4->CCR4 = (faze+665*3/4)%665;
	}
}
```

Налаштування переривання INTTIM_Config є аналогічним до попереднього прикладу та відрізняється лише частотою спрацювання:

```
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 = 400 - 1;
	tim_struct.TIM_Prescaler = SystemCoreClock/100000 - 1;
	tim_struct.TIM_ClockDivision = TIM_CKD_DIV1;
	tim_struct.TIM_CounterMode = TIM_CounterMode_Up;
	TIM_TimeBaseInit(TIM2, &tim_struct);
	TIM_ARRPreloadConfig(TIM2, ENABLE);
	TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
	TIM_Cmd(TIM2, ENABLE);
}
```

В результаті головна функція містить лише налаштування ШІМ сигналу та переривання для змін яскравості. Всі події генеруються апаратно та за допомогою переривання, в результаті цикл можна зайняти іншими корисними задачами:

```
int main(void)
{
	SystemInit();
	SystemCoreClockUpdate();
	LedsPWMInit();
	PWM_Init();
	INTTIM_Config();
	while(1){

	}
}
```

3.4. Використання аналогово-цифрового перетворювача

Мікроконтролер STM32F407VG включає в себе три АЦП. Розрядність всіх АЦП становить 12 біт. Кожен перетворювач здатний приймати сигнал з шістнадцяти зовнішніх каналів. Крім того, до складу контролера входить датчик температури. Діапазон вхідної напруги від датчику температури становить 1.8...3.6В. Датчик температури підключений до вхідного каналу ADC_IN16, який використовується для того, щоб перетворити вихідну напругу сенсора в цифрове значення. Внутрішній датчик температури призначений лише для

відстеження зміни температури, а не для її вимірювання, оскільки зсув показників датчика може змінюватися в ході змін параметрів процесу. Тому, якщо необхідно точне вимірювання абсолютних значень температури, то для цього краще використовувати зовнішній датчик, який призначений саме для точного вимірювання температур.

Для того, щоб правильно визначити подану напругу, необхідно провести додаткові вимірювання. Для цього за допомогою вольтметра визначаємо напругу, яка відповідає 0b111111111111=4095, підключивши вольтметер до відповідних виходів подачі живлення на процесор. Наприклад, якщо вольтметр показав, що в напруга дорівнює 2.96В, записуємо даний коефіцієнт безпосередньо в код програми.

Розрахунок напруги здійснюється за наступною формулою:

$$U = \frac{U_{ref} \cdot ADC}{ADC_{\max}},$$

де U_{ref} – виміряне значення напруги живлення;

ADC – показник аналогово-цифрового перетворювача;

ADC_{max} – максимально можливий показник аналогово-цифрового перетворювача, в нашому випадку для процесорів STM32F… має значення 4095.

Наступний програмний код демонструє використання цифрового вольтметру в режимі вимірювання по запиту. Звісно, існують режими автоматичного запису виміряних значень з кількох каналів по DMA до оперативної пам'яті в фоновому режимі, однак одноразове вимірювання дозволяє використати вимірювання в той час коли воно саме потрібно з більш простими налаштуваннями та використанням меншої кількості бібліотек:

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

void Led_init(){
```

```
	GPIO_InitTypeDef gpio;
	RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
	gpio.GPIO_OType = GPIO_OType_PP;
	gpio.GPIO_PuPd = GPIO_PuPd_NOPULL;
	gpio.GPIO_Mode = GPIO_Mode_OUT;
	gpio.GPIO_Pin = GPIO_Pin_12;
	gpio.GPIO_Speed = GPIO_Speed_100MHz;
	GPIO_Init(GPIOD, &gpio);
}

void Led_on(){
	GPIO_SetBits(GPIOD, GPIO_Pin_12);
}

void Led_off(){
	GPIO_ResetBits(GPIOD, GPIO_Pin_12);
}
```

Перші три наведені функції знайомі по попереднім прикладам і нічого нового не містять в своєму коді. Новою є наступна функція:

```
void Adc_gpio_init() {
	GPIO_InitTypeDef gpio;
	RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
	GPIO_StructInit(&gpio);
	gpio.GPIO_Pin = GPIO_Pin_0;
	gpio.GPIO_Mode = GPIO_Mode_AIN;//GPIO_Mode_AN
	GPIO_Init(GPIOA, &gpio);
}
```

Тут проводиться налаштування виводу мікросхеми на роботу як аналоговий вхід. Від відомих до цього налаштувань тут використано функцію заповнення структури налаштування на значення по замовчанню GPIO_StructInit, далі новим є лише режим роботи лапки мікросхеми як аналоговий вхід GPIO_Mode_AIN.

Наступна функція відповідає безпосередньо за налаштування цифрового вольтметру:

```
void Adc_init() {
	  ADC_InitTypeDef adc;
	  ADC_CommonInitTypeDef adc_init;
	  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
	  ADC_DeInit();

	  ADC_StructInit(&adc);
	  adc_init.ADC_Mode = ADC_Mode_Independent;
	  adc_init.ADC_Prescaler = ADC_Prescaler_Div8;

	  adc.ADC_ScanConvMode = ENABLE;
```

```
        adc.ADC_ContinuousConvMode = ENABLE;
        adc.ADC_ExternalTrigConv = ADC_ExternalTrigConvEdge_None;
        adc.ADC_DataAlign = ADC_DataAlign_Right;
        adc.ADC_Resolution = ADC_Resolution_12b;

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

Наведена функція використовує дві структури налаштувань ADC_InitTypeDef та ADC_CommonInitTypeDef. Далі розглянемо використані налаштування:

ADC_Mode = ADC_Mode_Independent – режим незалежних вимірювань. Перелік можливих режимів можна переглянути у файлі «stm32f4xx_adc.h» в розділі defgroup ADC_Common_mode.

ADC_Prescaler = ADC_Prescaler_Div8 – визначає тактову частоту надану всім АЦП, дільник частоти може мати значення 2, 4, 6 та 8.

ADC_DMAAccessMode – відповідає за використання одного з каналів DMA, в нашому випадку DMA не використовується, тому залишається нульове значення.

ADC_TwoSamplingDelay – цей параметр відповідає за паузу між послідовними вимірами в автоматичному режимі, тут використовуються окремі вимірювання, тому значення обрано по замовчанню ADC_TwoSamplingDelay_5Cycles=0.

ADC_Resolution = ADC_Resolution_12b – відповідає за налаштування режиму роздільної здатності АЦП. Цей параметр може приймати значення 6, 8, 10 та 12 бітів.

ADC_ScanConvMode = ENABLE або DISABLE – вказує, чи конвертація виконується в багатоканальному чи одноканальному режимі.

ADC_ContinuousConvMode = ENABLE або DISABLE – вказує чи виконується вимірювання в режимі безперервного або одиночного режиму.

ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None – визначає що зовнішні синхроімпульси вимірювання не використовуються.

ADC_ExternalTrigConv – відповідає за вибір джерела зовнішніх синхроімпульсів, джерелом можуть працювати таймери або зовнішній сигнал, який подається до мікроконтролеру.

ADC_DataAlign = ADC_DataAlign_Right – задає положення отриманих значень праворуч, тобто виміряне значення належить проміжку 0..4095.

ADC_NbrOfConversion – вказує кількість вимірювань АЦП які буде зроблено для регулярної групи каналів. Цей параметр має коливатися від 1 до 16.

Останні команди задіяють обрані налаштування та вмикають ADC1.

Наступна функція виконує безпосереднє вимірювання напруги на обраному каналі. До цього відповідний вихід мікросхеми повинен бути налаштований як аналоговий вхід:

```
u16 ReadADC1(u8 channel) {
```

```
        ADC_RegularChannelConfig(ADC1, channel, 1,
ADC_SampleTime_3Cycles);
        ADC_SoftwareStartConv(ADC1);
        while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);
        return ADC_GetConversionValue(ADC1);
}
```

Функція містить вибір каналу для вимірювання де вказано, що вимірювання здійснює ADC1, вказується канал а потім їх кількість, в разі групового вимірювання кількість буде більшою за одиницю. Останнім параметром вказано час вимірювання в циклах (деякі значення з проміжку від 3 до 480).

Наступною командою проводиться запуск вимірювання. Далі в циклі очікується прапор завершення вимірювання, після чого виміряне значення повертається з функції.

Наступна функція вже відома за минулими прикладами:

```
void Delay(volatile int Val) {
      while(Val--);
}
```

Тіло програми містить функції налаштувань, після яких в головному циклі проводиться вимірювання напруги. Виміряне значення використане для задання шпаруватості ШІМ сигналу, який подано на вбудований на плату зелений світлодіод – яскравість світлодіоду пропорційна поданій на вхід напрузі:

```
int main(void) {
      Adc_gpio_init();
      Adc_init();
      Led_init();
      unsigned int bin_code=0;
      float voltage=0.0f;
      while(1){
            bin_code = ReadADC1(ADC_Channel_0);
            voltage = bin_code * 2.96 / 0b111111111111;
            /* voltage debug control only */
            Led_on();
            Delay(bin_code);
            Led_off();
            Delay(0b111111111111 - bin_code);
      }
}
```

Тут можна модернізувати програмний код для використання апаратного генерування ШІМ сигналу, але залишимо цю задачу для читача.

3.5. Використання універсального асинхронно-синхронного приймача-передатчику USART

USART – універсальний асинхронно-синхронного приймач-передатчик, пристрій для передачі інформації через послідовний порт, коли біти числа передаються по черзі по одній лінії. Мікроконтролер використовує два виходи, один на приймання інформації, другий – передачу. Кожен пристрій на лінії передачі утримує високий рівень. За цією ознакою пристрій може визначати, чи активний пристрій на іншій стороні та відстежити момент його підключення.

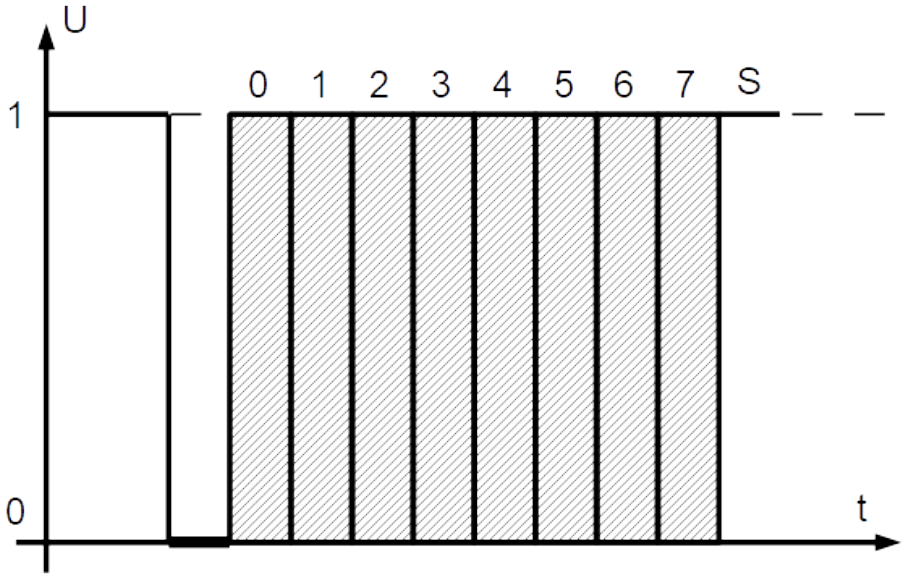


Рис. 3.6. Передача восьми розрядного байту з одним стоп-бітом без контролю парності

Діаграма передачі байту показана на рис. 3.6. З діаграми видно, для того щоб приймач зрозумів, що почалася передача байту, передавач повинен видати низький рівень на час передачі одного біту. Далі передається байт від молодшого до старшого біту, але порядок може бути й зворотнім. В останню чергу передається біт високого рівня для гарантії, що стартовий біт наступного байту не буде пропущений приймачем.

На діаграмі показано мінімальна конфігурація пакету сигналів для передачі байту. Для більш надійного зв’язку можна використовувати перед стоп-бітом ще один додатковий біт парності. За домовленістю цей біт парності

гарантує парну або непарну кількість одиничних бітів в пакеті. Тим самим система може перевіряти випадки випадкового помилкового прийняття біту, при цьому парність бітів в доповненому байті порушиться і приймач матиме ознаку прийняття інформації з помилкою. Також деякі повільні пристрої вимагають більш довгого стоп-біту довжиною 1,5 або 2 від довжини імпульсу на один біт. Алгоритми передачі та прийому байту показано на рис. 3.7 та 3.8.

За логікою цей протокол повністю відповідає комп'ютерному COM порту, тому за цим протоколом через мікросхему (наприклад max232) перетворювача рівнів, мікросхему можна під'єднати до комп'ютеру.

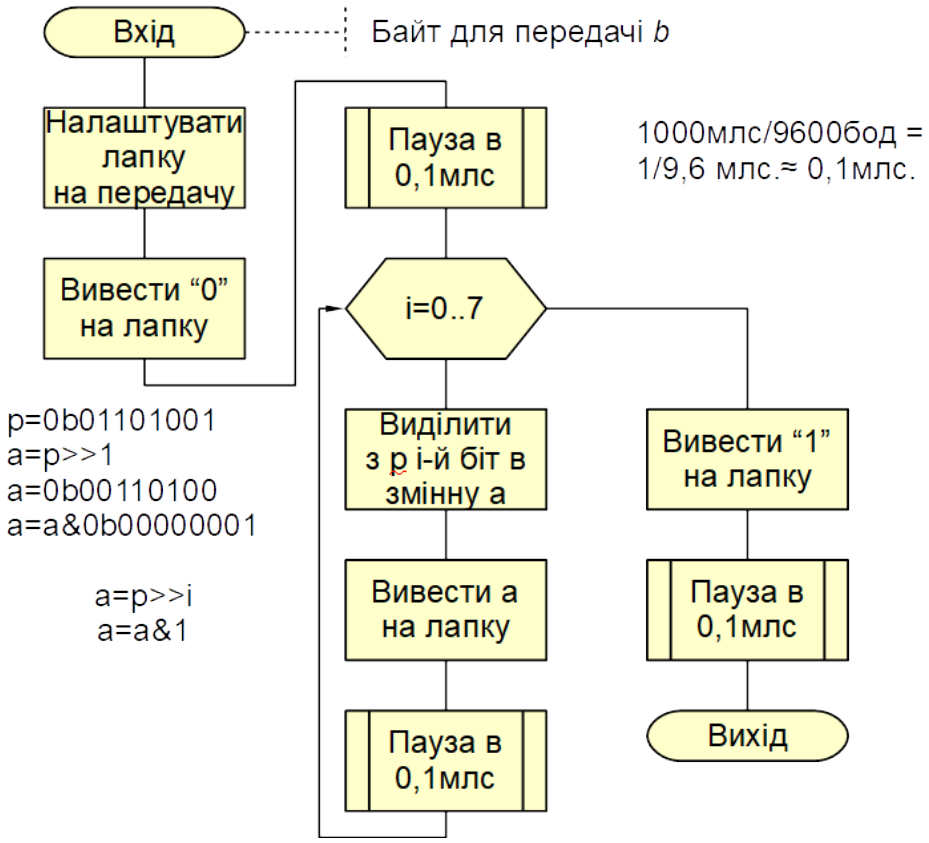


Рис. 3.7. Алгоритм передачі байту в послідовний USART порт

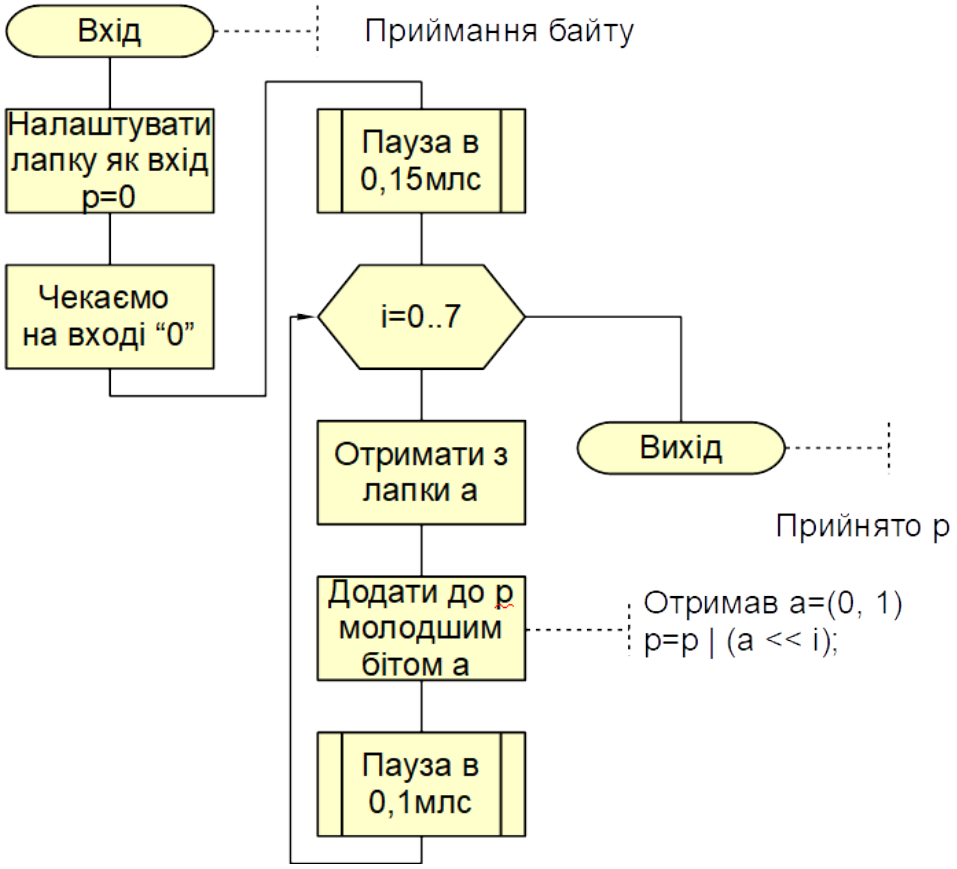


Рис. 3.8. Алгоритм прийому байту з послідовного USART порту

В мікроконтролері реалізована апаратна реалізація послідовного синхронно-асинхронного інтерфейсу. Розглянемо програму, яка реалізує приймання та передачу байтів по USART інтерфейсу.

Для використання апаратного USART інтерфейсу будуть потрібні наступні бібліотеки:

```
#include <stm32f4xx.h>
#include <stm32f4xx_rcc.h>
#include <stm32f4xx_gpio.h>
#include <stm32f4xx_usart.h>
#include <misc.h>
```

Більшість з них відомі, новою тут буде бібліотека обслуговування послідовного порту USART. Далі йде функція для налаштування виходів мікросхеми на альтернативні функції. З документації можна визначити, що USART1 використовує 9 та 10 піни порту A:

```
void Usart_gpio_init(){
	GPIO_InitTypeDef gpio;
	RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
	RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
```

```
    gpio.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10;
    gpio.GPIO_Mode = GPIO_Mode_AF;
    gpio.GPIO_Speed = GPIO_Speed_50MHz;
    gpio.GPIO_OType = GPIO_OType_PP;
    gpio.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_Init(GPIOA, &gpio);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource9, GPIO_AF_USART1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource10, GPIO_AF_USART1);
}
```

Наступним кроком є налаштування самого інтерфейсу USART1 на приймання-передачу байтів:

```
void Usart_init(){
    USART_InitTypeDef usart;
    usart.USART_BaudRate = 9600;
    usart.USART_WordLength = USART_WordLength_8b;
    usart.USART_StopBits = USART_StopBits_1;
    usart.USART_Parity = USART_Parity_No;
    usart.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    usart.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
    USART_Init(USART1, &usart);
    USART_Cmd(USART1, ENABLE);
}
```

Тут використано структуру USART_InitTypeDef. Розглянемо використані параметри. USART_BaudRate – вказується швидкість передачі в бодах (кількість бітів за секунду), тут обрано стандартне значення 9600 бодів, що відповідає налаштуванню по замовчанню Bluetooth модуля HC-06, який авторами був використаний для пробного зв’язку з комп’ютером. Далі USART_WordLength відповідає за довжину байту, який передається. При передачі між контролерами це поле може набувати значень USART_WordLength_8b та USART_WordLength_9b. Однак при обміні дев’ятирозрядні байти при зв’язку з комп’ютером не використовуються.

USART_StopBits – може набувати значення USART_StopBits_0_5, USART_StopBits_1, USART_StopBits_1_5 та USART_StopBits_2, що відповідає проміжку між байтами в 0.5, 1, 1.5 та 2 одиничних біти. Ці проміжки використовують для гарантії часового проміжку між двома бітами, також довгі проміжки 1.5 та 2 біти можуть бути задіяні при обміні інформацією з повільними пристроями – за цей час вони повинні встигнути обробити прийнятий щойно байт та підготуватися до приймання наступного.

USART_Parity – використовується для контролю правильності приймання байту. Значення USART_Parity_Even означає використання додаткового біту, який гарантує парну кількість одиничних бітів, якщо буде помилка прийнятті байту, парність бітів порушиться; USART_Parity_Odd – кількість одиничних бітів є непарною; USART_Parity_No – біт парності не використовується. Біт парності не використовують, коли помилка в передачі допустима або контролюється правильність передачі іншими програмними засобами, наприклад пакет байтів доповнюють контрольною сумою та перевіряють прийняту контрольну суму та розраховану самостійно.

USART_HardwareFlowControl – апаратний контроль дозволу на приймання та передачі байтів. Може набувати наступних значень: USART_HardwareFlowControl_None, керування дозволом на приймання та передачу не використовується і передача/прийом може відбуватися в довільний момент часу; USART_HardwareFlowControl_RTS, використовується окремий вихід мікросхеми, на якому надається сигнал пристрою-передатчику, що данні можна/заборонено надсилати; USART_HardwareFlowControl_CTS, використовується окремий вхід мікросхеми, на якому приймається сигнал пристрою-приймача, що данні можна/заборонено надсилати; USART_HardwareFlowControl_RTS_CTS – використано двонаправлене апаратне керування дозволом на передачу та прийом даних.

Код USART_Mode = USART_Mode_Rx | USART_Mode_Tx відповідає за налаштування USART в режим приймання та передачі інформації. Також можливі варіанти, коли мікроконтролер лише приймає або лише передає інформацію.

Наступна функція відповідає за налаштування переривання подій від USART пристрою:

```
void Usart_irq_init(){
	NVIC_InitTypeDef irqconf;
	USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
	irqconf.NVIC_IRQChannel = USART1_IRQn;
	irqconf.NVIC_IRQChannelPreemptionPriority = 0;
	irqconf.NVIC_IRQChannelSubPriority = 0;
	irqconf.NVIC_IRQChannelCmd = ENABLE;
```

```
    NVIC_Init(&irqconf);
}
```

Цей код мало чим відрізняється від попереднього налаштування переривання від таймеру, а наступний код вам відомий по керуванню світлодіодами:

```
void Led_init(){
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
    GPIO_InitTypeDef gpio;
    gpio.GPIO_Mode = GPIO_Mode_OUT;
    gpio.GPIO_OType = GPIO_OType_PP;
    gpio.GPIO_Pin = GPIO_Pin_12;
    gpio.GPIO_PuPd = GPIO_PuPd_NOPULL;
    gpio.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_Init(GPIOD, &gpio);
}
void Led_on(){
    GPIO_SetBits(GPIOD, GPIO_Pin_12);
}
void Led_off(){
    GPIO_ResetBits(GPIOD, GPIO_Pin_12);
}
```

Наступна функція є обробником переривання подій від USART пристрою, при чому для всіх подій призначений лише один обробник. Тому, якщо в налаштуванні переривань вказано кілька подій в якості джерела переривання, функція викликатиметься для кожної з подій. Для визначення події, яка саме стала причиною переривання потрібно перевіряти регістр стану USART пристрою за допомогою функції USART_GetITStatus. В нашому випадку використано подію наявності нового прийнятого байту USART_IT_RXNE:

```
volatile unsigned char lastByte = 0;
void USART1_IRQHandler(void){
    if( USART_GetITStatus(USART1, USART_IT_RXNE) ){
        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
        lastByte = USART_ReceiveData(USART1);
        USART_SendData(USART1, lastByte); //echo
    }
}
```

По визначенню події потрібно обов’язково очистити відповідний біт-ознаку події, інакше система вважатиме, що подія не оброблена і виклику переривання після надходження наступного байту не буде, а наступні байти взагалі будуть проігноровані.

Наступна функція використовується для генерування пауз:

```
void Delay(volatile int Val) {
	while(Val--);
}
```

Головна ж функція викликає процедури налаштувань, при цьому світлодіод за програмно генерованим ШІМ сигналом, змінює яскравість світіння:

```
int main(void) {
	SystemInit();
	SystemCoreClockUpdate();
	Led_init();
	Usart_gpio_init();
	Usart_irq_init();
	Usart_init();
	while(1){
		Led_on(LED0);
		Delay(0x0100*lastByte);
		Led_off(LED0);
		Delay(0x0100*(255-lastByte));
	}
}
```

В якості для завдання для вдосконалення власного розуміння роботи вбудованих пристроїв при обміні інформацією, спробуйте реалізувати зміну яскравості світіння чотирьох світлодіодів, де кожен з світлодіодів отримує власні параметри світіння.

3.6. Робота з SPI

Serial Peripheral Interface (SPI) - популярний інтерфейс для послідовного обміну даними між мікросхемами. Шина SPI організована за принципом "майстер-підлеглий". В якості майстра шини зазвичай виступає мікроконтролер, але їм також може бути програмована логіка, DSP-контролер або спеціалізована інтегральна схема. До одного керуючого пристрою паралельно включають ланцюг підлеглих пристроїв (рис. 3.10).

Головним складовим блоком інтерфейсу SPI є звичайний регістр зсуву, сигнали синхронізації і введення/виводу бітового потоку з якого і утворюють інтерфейсні сигнали. Таким чином, протокол SPI правильніше що назвати не

протоколом передачі даних, а протоколом обміну даними між двома зсувними регістрами, кожен з яких одночасно виконує і функцію приймача, і функцію передавача. Неодмінною умовою передачі даних по шині SPI є генерація сигналу синхронізації шини. Цей сигнал має право генерувати лише майстер шини і від цього сигналу повністю залежить робота всіх підлеглих пристроїв на шині.

Приклад найбільш простого підключення по шині SPI показаний на малюнку 3.9 та 3.10, де однойменні висновки з'єднуються між собою. Присутні два канали передачі даних (MOSI і MISO) один канал для подачі тактових імпульсів (SCLK), а також лінія включення підлеглого пристрою (SS). Підлеглий стає активним при подачі низького рівня по лінії SS.

Плата STM32F4Discovery має в своєму складі мікросхемний акселерометр (рис. 3.11), який може працювати в режимах обміну даними I2C та SPI. Використовуватимемо останній, бо розводка плати виконана саме під цей режим.

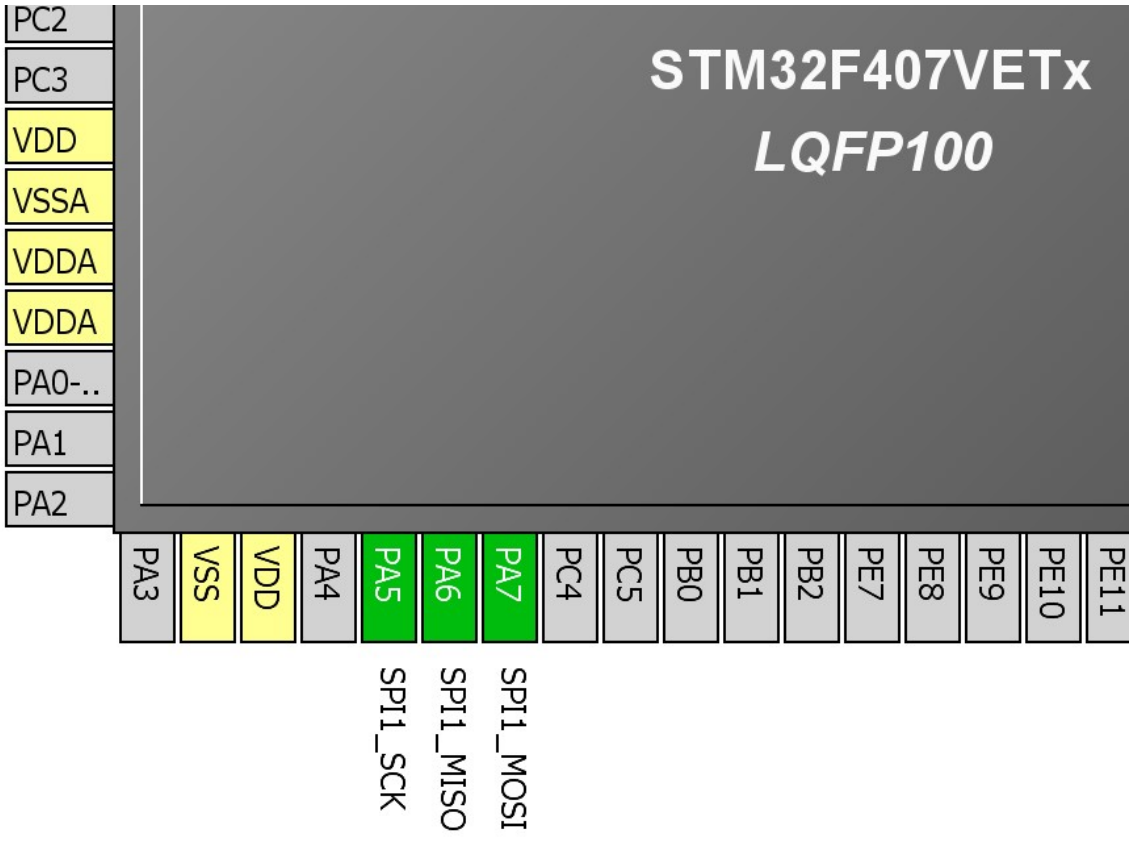


Рис. 3.9. Позначення виходів SPI1 в STM32CubeMX

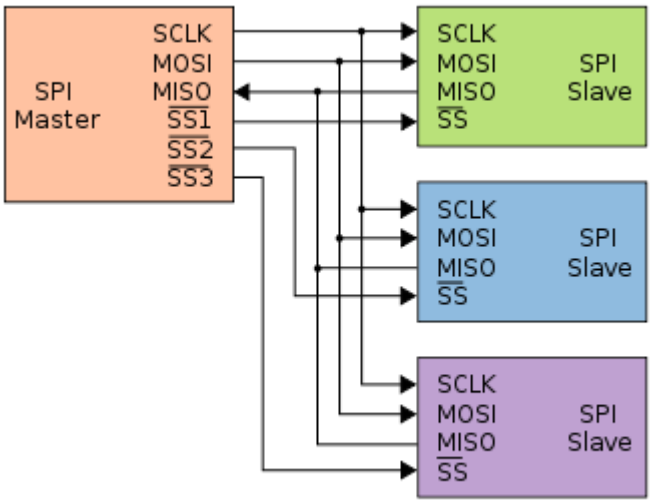


Рис. 3.10. Схема включення «один ведучий – кілька незалежних підлеглих» [https://uk.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:SPI_three_slaves_daisy_chained.svg]

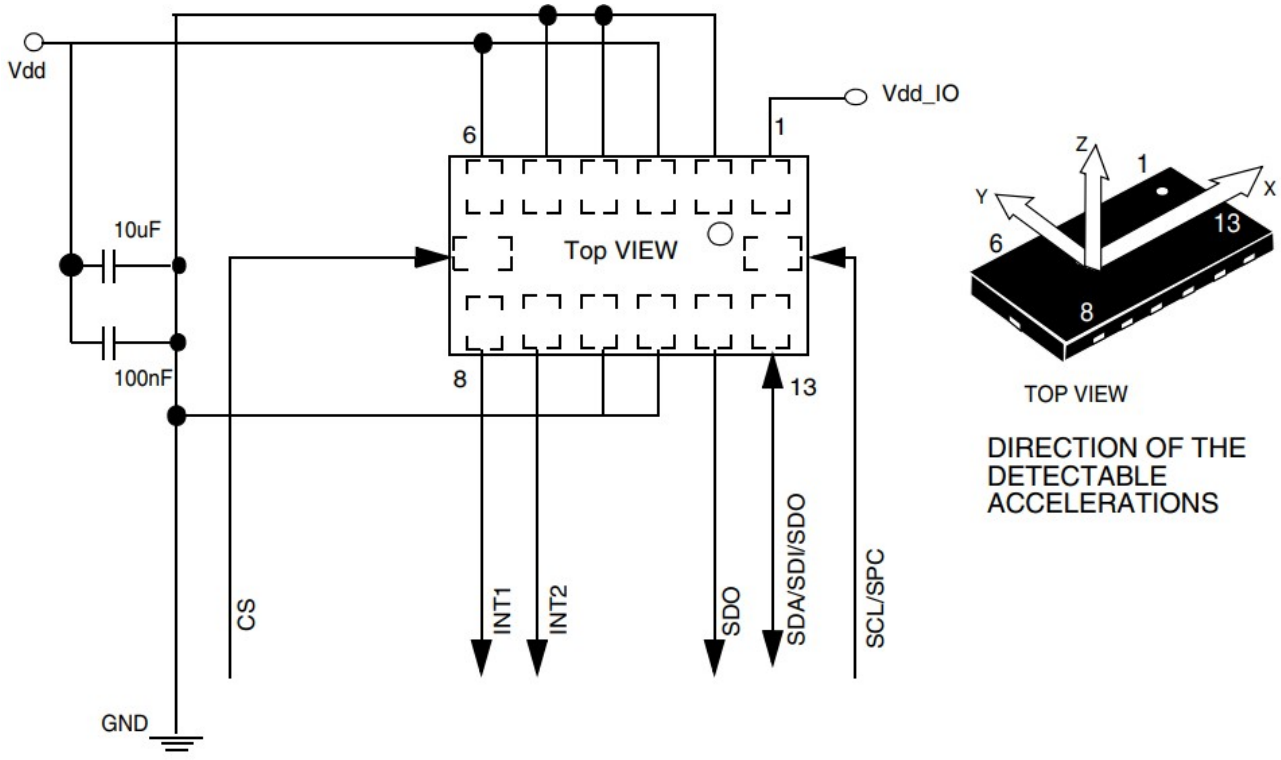


Рис. 3.11. Схема під’єднання акселерометру з офіційної документації

На рис. 3.11 показано контакти мікросхеми акселерометру. З причини використання SPI, виходи мікросхеми мають наступну відповідність:

SDA=MOSI, SCL=SCK, SDO=MISO. Вибір інтерфейсу вибирається на вході мікросхеми CS. Якщо на нозі логічний 1, то акселерометр використовує I2C, в іншому випадку – SPI. Також з документації маємо наступні характеристики мікросхеми:

1) Напруга живлення 2.16-3.6 В.
2) Вимірювання прискорення по трьох осях.
3) Два діапазону вимірювання 2G / 8G.
4) Два настроюються виходу для переривань.
5) Самодіагностика.
6) Виявлення кліків (постукувань).
7) Вбудований фільтр.

Розглянемо порядок роботи інтерфейсу.

Взаємодія з акселерометром здійснюється через його регістри. Щоб прочитати або щось записати в них дані, потрібно відправити їх через SPI посилку певного формату. В документації наведено датаграми обміну інформацією (рис. 3.12).

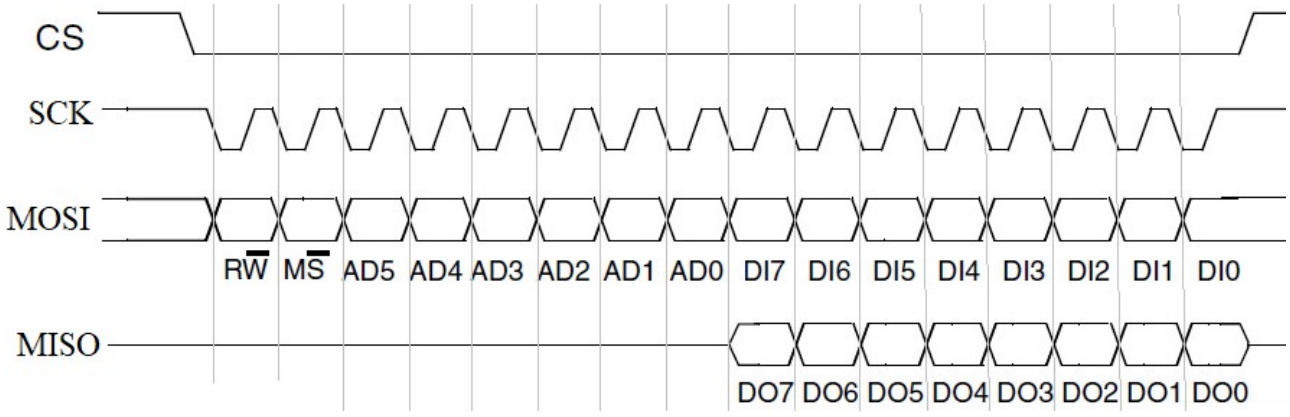


Рис. 3.12. Датаграма процесу обміном байтом майстра з підлеглим пристроєм

DO7..DO0 – байт даних відправлений акселерометром в мікроконтролер;

DI7..DI0 – байт даних переданий мікро контролером в акселерометр;

AD5 .. AD0 – адреса яка записується/зчитується до/з регістра;

RW – якщо біт = 0 то байт даних DI7..DI0 буде записаний в регістр за

адресою AD5..AD0, в іншому випадку (RW=1) байт даних DO7..DO0 буде прочитаний з регістру за адресою AD5..AD0.

MS - використовується якщо ми хочемо прочитати/записати кілька регістрів поспіль. Якщо біт скинутий (=0), то після передачі адреси ми будемо зчитувати/записувати один і той же регістр незалежно від того, скільки разів буде зроблена спроба читання/запису. Якщо ж цей біт встановлений (=1), то адреса буде автоматично збільшуватися на одиницю після кожного запису або читання, це зручно використовувати для запису/зчитування кілька байтів поспіль.

Для того щоб прочитати вміст одного регістра (наприклад з адресою 0x0F) потрібно виконати наступну послідовність дій:

1) Вихід CS притиснути до землі (=0).
2) Передати перший байт (0x8F) який містить адресу регістра який потрібно зчитати; при цьому скинутий біт MS (так як ми читаємо тільки один регістр) і встановлений біт RW (адже ми читаємо дані), що й спричинило доповнення адреси до значення 0x8F.
3) Передати довільний байт. Адже інтерфейс SPI так влаштований, що підлеглий пристрій не може передавати синхронізуючі імпульси. Тому акселерометр почне передавати значення з регістру лише в момент передачі мікроконтролером другого байту (який по суті буде просто проігноровано).
4) Прочитати значення з регістра даних SPI (те, що прийшло від акселерометра).
5) Встановити високий логічний рівень на виводі CS.

Якщо потрібно швидко читати один і той самий регістр, то можна постійно повторювати кроки 3 і 4.

Запис одного регістра не складніше за читання, спробуємо записати байт 0x47 в регістр за адресою 0x20. Для цього потрібно:

1) Вихід CS притиснути до землі.
2) Передати перший байт (0x20) який містить адресу регістра для

запису, скинутих біт MS=0 (так записується лише один регістр) і скинутий біт RW=0 (пишемо дані).

3) Передати нове значення регістра (0x47)
4) Встановити високий логічний рівень на виводі CS

Якщо є потреба швидко писати будь-які значення в один і той же регістр, то можна постійно повторювати 3-й крок.

Читання кількох регістрів виконують за наступним алгоритмом:

1) Вихід CS притиснути до землі.
2) Передати перший байт (0xCF) який містить адресу початкового регістру для читання, та у якому встановлений біт MS (читання кількох регістрів), та встановлений біт RW (ознака читання даних).
3) Передати довільний байт (який по суті буде просто проігноровано).
4) Прочитати значення з регістра даних SPI (відповідь від акселерометра).
5) Повторювати кроки 3 і 4 поки не прочитаємо потрібну кількість регістрів. Адреса читання буде збільшуватися на одиницю самостійно.
6) Встановити високий логічний рівень на виводі CS.

І нарешті, запис кількох регістрів виконують за наступним алгоритмом:

1) Вихід CS притиснути до землі.
2) Передати перший байт (0x4F) який містить адресу початкового регістру для читання, та у якому встановлений біт MS (читання/запис кількох регістрів), та скинутий біт RW (ознака запису даних).
3) Передати значення до регістру 0x0F.
4) Передати значення до регістру 0x10.
5) Передати значення до регістру 0x11.
6) …
7) Встановити високий логічний рівень на виводі CS.

Залишається з’ясувати, які саме регістри читати та записувати при роботі

з пристроями. Цю інформацію можна отримати для конкретного пристрою з його документації. Для акселерометру розглянемо документовані регістри та параметри.

Who_Am_I (0x0F) – регістр містить ідентифікатор акселерометру LIS302DL. Значення можна лише прочитати, прочитане значення повинне бути рівним 0x3b. За цим значенням можна перевіряти наявність зв'язку з акселерометром.

CTRL_REG1 (0x20) – перший регістр стану акселерометру. Його біти відображені нижче, де в дужках наведене значення по замовчанню:

DR (0), PD (0), FS (0), STP (0), STM (0), Zen (1), Yen (1), Xen (1).

DR – біт налаштовує частоту вибірки. Якщо він встановлений, то частота 400Гц, якщо скинуто то 100Гц. Впливає на точність вимірювань.

PD – біт керування живленням. Біт скинутий – живлення вимкнене і навпаки. Поки не встановлено цей біт вимірювання не проводяться.

FS – біт вибору діапазону вимірювань. Біт скинутий – діапазон ±2g, якщо ж встановлений, то діапазон вимірювань складає ±8g.

STP і STM – біти керування режимом самодіагностики.

Zen, Yen, Xen – біти дозволу генерування сигналу готовності даних для осі Z, Y та X. Якщо скинути біт, прискорення по цій осі буде вимірюватися, але прапор готовності даних (в регістрі STATUS_REG) встановлено не буде.

CTRL_REG2 (0x21) – другий регістр налаштування:
SIM (0), BOOT (0), - , FDS (0), HP_FF_W-U2 (0), HP_FF_W-U1 (0), HP_coeff2 (0), HP_coeff1 (0).

SIM – біт вибору режиму SPI. Якщо встановлено, то акселерометр переключиться в трьохпровідний режим SPI (прийом і передача буде відбуватися по одному дроту). За замовчуванням він скинутий і SPI працює в звичайному 4-х дротовому режимі.

BOOT – при запису одиниці в цей біт, відбувається скидання всіх налаштувань акселерометра за замовчуванням. Регістри статусу не скидаються.

FDS – включити/вимкнути фільтр. Якщо біт встановлено, то тяжіння землі

не робить вплив на вимірювання і навпаки. Простіше кажучи, якщо фільтр включений, то поки до акселерометру не надаватимуть прискорення, в регістрах даних X, Y, Z будуть нулі (або майже нулі). Якщо ж вимкнути фільтр, то дані в регістрах X, Y, Z будуть залежати від орієнтації акселерометра в просторі.

HP_FF_WU2 і HP_FF_WU1 – включають/вимикають фільтр верхніх частот для двох модулів генеруючих переривання (FF_WU1 і FF_WU2). Частота зрізу фільтра налаштовується за допомогою бітів HP_ coeff2 і HP_ coeff1:

Табл. 3.1 – Вибір частоти зрізу для фільтру

HP_ coeff2	HP_ coeff1	Частота зрізу (Біт DR = 0)	Частота зрізу (Біт DR = 1)
0	0	2 Гц	8 Гц
0	1	1 Гц	4 Гц
1	0	0.5 Гц	2 Гц
1	1	0.25 Гц	1 Гц

CTRL_REG3 (0x21) – Регістр налаштування переривань:

IHL (0), PP_OD (0), I2CFG2 (0), I2CFG1 (0), I2CFG0 (0), I1CFG2 (0), I1CFG1 (0), I1CFG0 (0).

IHL – якщо біт скинутий, то при виникненні переривання, на ногах INT1 і INT2 з’явиться логічна одиниця. Якщо біт встановлено, то поки переривання не сталося на висновках INT1 і INT2 присутня логічна одиниця, а в момент виникнення переривання, на вихід подається логічний нуль.

PP_ OD – тип виходів INT1 та INT2. 0 – підтяжка до нуля, 1 – відкритий колектор.

За допомогою бітів I2 CFG [2..0] і I1 CFG [2..0] можна вибрати джерело викликів переривання INT1 і INT2:

Табл. 3.2 – Вибір джерела переривання

I1 (2) _CFG2	I1 (2) _CFG1	I1 (2) _CFG0	Джерело переривання
0	0	0	Переривання вимкнені, виходи підтягнені до землі.
0	0	1	FF_WU_1
0	1	0	FF_WU_2
0	1	1	FF_WU_1 та FF_WU_2
1	0	0	Дані готові.
1	1	1	Переривання від «кліка».

FF_WU_1 і FF_WU_2 це два незалежних блоку призначених для генерації переривань в разі якщо прискорення по одній або декількох осях, вийде за заданий поріг.

HP_FILTER_RESET (0x23) –Якщо буде включений фільтр верхніх частот, після читання вмісту цього регістра в регістри Out_X, Out_Y і Out_Z запишуться нулі. Тут важливий сам факт читання регістра, а не ті дані, які прочиталися з нього.

STATUS_REG (0x27) – регістр статусу, кожен біт якого є ознакою певної події: ZXYOR (0), ZOR (0), YOR (0), XOR (0), ZYXDA (0), ZDA (0), YDA (0), ZDA (0).

ZXYOR - Дані в регістрах Out_X, Out_Y і Out_Z були переписані новими значеннями, а їх попередній вміст не встигли прочитати.

ZOR - Дані в регістрі Out_Z переписано новими, до того як були прочитані попередні.

YOR - Дані в регістрі Out_Y переписано новими, до того як були прочитані попередні.

XOR - Дані в регістрі Out_X переписано новими, до того як були прочитані попередні.

ZYXDA - Доступні нові дані в регістрах Out_X, Out_Y і Out_Z.

ZDA - Доступні нові дані в регістрі Out_Z.

YDA - Доступні нові дані в регістрі Out_Y.

XDA - Доступні нові дані в регістрі Out_X.

OUT_X (0x29) – восьмирозрядний регістр даних, в який записується прискорення по осі X.

OUT_Y (0x2b) – восьмирозрядний регістр даних, в який записується прискорення по осі Y.

OUT_Z (0x2d) – восьмирозрядний регістр даних, в який записується прискорення по осі Z

У акселерометра є два незалежних модуля, які вміють самостійно читати дані з регістрів Out_X , Out_Y , Out_Z і на їх основі генерувати переривання. Для налаштування кожного модуля використовуються 4 регістра які будуть описані нижче. Для зручності налагодження можна причепити до ніг INT1/INT2 по світлодіоду через резистор. Тоді якщо переривання трапиться це буде відразу видно.

FF_WU_CFG_1 (0x30), FF_WU_CFG_2 (0x34) – регістр налаштовує умови виникнення переривання.

AOI (0), LIR (0), ZHIE (0), ZLIE (0), YHIE (0), YLIE (0), XHIE (0), XLIE (0)

AOI – біт визначає в коли згенерувати переривання. Якщо він скинутий, то переривання виникне, коли відбудуться всі події обраних бітами XHIE/ YHIE/ ZHIE/ XLIE/ YLIE/ ZLIE.

LIR – якщо біт варто, то коли ми будемо читати регістр FF_WU_SRC_1 (2), його вміст автоматично скидатиметься.

XHIE / YHIE / ZHIE - якщо біт виставлений, то акселерометр починає стежити за станом відповідного регістра даних, і коли значення прискорення за відповідною осі перевищить граничне – буде виставлений прапор в регістрі FF_WU_SRC_1 (2).

XLIE / YLIE / ZLIE - якщо біт виставлений, то акселерометр починає стежити за станом відповідного регістра даних, і коли значення прискорення за відповідною осі стане менше порогового – буде виставлений прапор в регістрі FF_WU_SRC_1 (2).

FF_WU_SRC_1 (0x31), FF_WU_SRC_2 (0x35) – регістр прапорів. Якщо біт LIR (в регістрі FF_WU_CFG_X) встановлено, то при зчитуванні цього регістра, він скидається. Якщо до цього відбулося переривання, то відразу після читання, обидві ноги INT1 / INT2 перемикаються в початковий стан.

-, IA (0), ZH (0), ZL (0), YH (0), YL (0), XH (0), XL (0).

IA - якщо біт встановлений, то переривання сталося.

ZH / YH / XH - якщо біт встановлений то прискорення за відповідною осі перевищило поріг.

ZL / YL / XL - якщо біт встановлений то прискорення за відповідною осі стало менше порога.

Природно, перші шість біт будуть встановлюватися лише в тому випадку, якщо встановлено біти XHIE / YHIE / ZHIE / XLIE / YLIE / ZLIE в регістрі FF_WU_CFG_1 (2).

Налаштування для генерування «кліків / постукувань» можна прочитати в документації до акселерометра.

На платі розробника акселерометр підключено за наступною схемою (рис. 3.13):

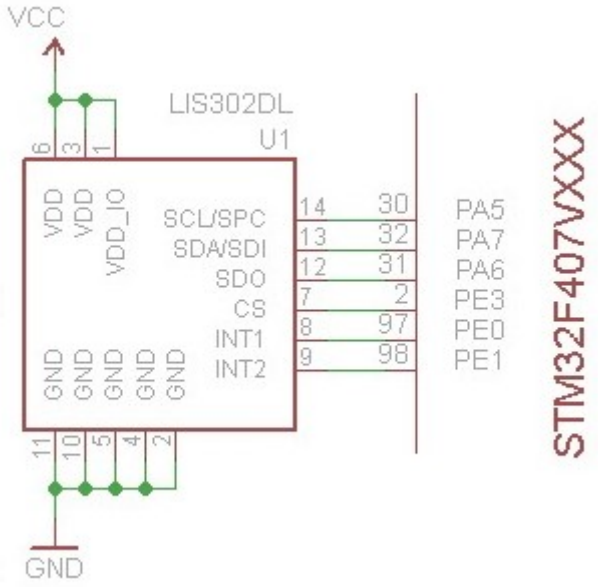


Рис. 3.13. Підключення акселерометру до мікроконтролера по шині SPI

В нашому випадку переривання використовуватися не будуть, задача

полягає в опитуванні показань прискорення та засвічування найвищого світлодіоду. Для цього потрібно в циклі читати регістр стану акселерометру STATUS_REG (0x27) і, при наявності встановленого прапору ZYXDA прочитати показання OUT_X (0x29) та OUT_Y (0x2b) – значення прискорення по двом напрямкам. Максимальне по модулю значення вказує пару протилежних світлодіодів, а знак – який саме світлодіод з пари потрібно включити.

Почнемо з ініціалізації SPI інтерфейсу:

```
void SpiInit() {
    GPIO_InitTypeDef GPIO_InitStructure;
    SPI_InitTypeDef SPI_InitStructure;

    // Тактування SPI1 та порту A
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1,
ENABLE);
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA |
RCC_AHB1Periph_GPIOE, ENABLE);

    // Налаштування виходів для роботи в SPI1
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource7,
GPIO_AF_SPI1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource5,
GPIO_AF_SPI1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource6,
GPIO_AF_SPI1);
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed =
GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd =
GPIO_PuPd_NOPULL;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7 |
GPIO_Pin_6 | GPIO_Pin_5;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
```

```
    GPIO_InitStructure.GPIO_Speed =
GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
    GPIO_Init(GPIOE, &GPIO_InitStructure);
    GPIO_SetBits(GPIOE,GPIO_Pin_3); //Вихід CS

    //Заповнюємо структуру з налаштуванням SPI
    SPI_InitStructure.SPI_Direction =
SPI_Direction_2Lines_FullDuplex; //Двонаправлений
обмін
    SPI_InitStructure.SPI_DataSize =
SPI_DataSize_8b; //Восьмирозрядний режим
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
//Біт повинен прийматися в момент переходу сигналу
тактування від низького до високого рівня (рис.
3.12)
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft; //
Керування NSS сигналом відбувається програмно
    SPI_InitStructure.SPI_BaudRatePrescaler =
SPI_BaudRatePrescaler_32; //Преддільник SCK
    SPI_InitStructure.SPI_FirstBit =
SPI_FirstBit_MSB; // В байті може передаватися
першим старший або молодший біт, правильний напрям
визначається з документації пристрою, тут
встановлена передача зі старшого біту
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
// Режим - майстер
    SPI_Init(SPI1, &SPI_InitStructure);
//Застосовуємо налаштування SPI1
    SPI_Cmd(SPI1, ENABLE); // Вмикаємо модуль
    // Сигнал NSS контролюється програмно,
потрібно встановити його в одиницю, якщо ж його
скинути в нуль, то SPI перейде в мультимайстерну
топологію (сигнал скасування повноважень майстра).
    SPI_NSSInternalSoftwareConfig(SPI1,
SPI_NSSInternalSoft_Set);
}
```

Обмін інформацією будемо використовувати лише по одному байту. Передача/прийом буферу читач може реалізувати самостійно. Першочергово створимо функцію запису й одночасного приймання байту:

```
int WriteSPIData(uint8_t data) {
    while(SPI_I2S_GetFlagStatus(SPI1,
SPI_I2S_FLAG_BSY) == SET); //Чекаємо звільнення
пристрою
    SPI_I2S_SendData(SPI1, data); //Ініціюємо
передачу байту
    while(SPI_I2S_GetFlagStatus(SPI1,
SPI_I2S_FLAG_BSY) == SET); //Чекаємо завершення
передачі
    return SPI_I2S_ReceiveData(SPI1); //Повертаємо
вхідний байт
}
```

Тепер розглянемо запис байту до регістру за розглянутим до цього алгоритмом:

```
void SetReg(uint8_t address, uint8_t value) {
    GPIO_ResetBits(GPIOE,GPIO_Pin_3); //SS
встановимо в 0
    WriteSPIData(address); //Пишемо адресу
регістра
    WriteSPIData(value); //Відправляємо значення
до регістру
    GPIO_SetBits(GPIOE, GPIO_Pin_3); //SS
встановлюємо в 1 – обмін завершено
}
```

Тепер читання значення регістру:

```
int GetReg(uint8_t address) {
    int data=0; //Змінна для збереження відповіді
    address|=(1<<7); //Встановлюємо прапор RW=1 –
режим читання однобайтного читання
    GPIO_ResetBits(GPIOE,GPIO_Pin_3); //SS
встановимо в 0 – початок обміну
    WriteSPIData(address); //Пишемо адресу
регістра
```

```
    data = WriteSPIData(0x00); //Відправляємо
довільне значення до регістру, автоматично
отримаємо відповідь мікроконтролера
    GPIO_SetBits(GPIOE,GPIO_Pin_3); //SS
встановлюємо в 1 - обмін завершено
    return data; //Повертаємо отриманий результат
}
```

Тепер наведемо реалізацію головної функції програми:

```
int main(void)
{
    SystemInit();// Налаштування таймінгу
    SystemCoreClockUpdate();// Оновлення фактичної
частоти процесора
    LedConfig();//Ініціюємо світлодіоди
    SpiInit();//Ініціюємо обмін з акселерометром

    unsigned char idf = GetReg(0x0F); //Читаємо
ідентифікаційний байт пристрою
    if(idf!=0x3B){
        //Не наш пристрій, покажемо це
світлодіодами
        LedOn(LED1 | LED2 | LED3 | LED4);
        while(1); //Працювати нема з чим
    }
    SetReg(0x20, 0b11000000);//DR=1 (400Hz), PD=1
(живлення увімкнене), ініціювання акселерометру
    while(1){ //Робочий цикл:
        int x = GetReg(0x29); //Читаємо
прискорення по X
        int y = GetReg(0x2B); // - по Y
        if( Abs(x)>Abs(y) ){ //Обираємо світлодіод
            if(x>0){
                LedOff(LED2 | LED3 | LED4);
                LedOn(LED1);
            }else{
                LedOff(LED2 | LED1 | LED4);
                LedOn(LED3);
            }
        }else{
```

```
                if(y>0){
                    LedOff(LED1 | LED3 | LED4);
                    LedOn(LED2);
                }else{
                    LedOff(LED2 | LED1 | LED3);
                    LedOn(LED4);
                }
            }
        }
}
```

Для самостійного тренування реалізуйте самостійно роботу з пересиланням та прийманням групи байтів.

3.7. Використання DMA

Недоліком такого обміну є зайнятість контролеру під час обміну інформацією. Тому потрібно вміти передавати дані в автоматичному фоновому режимі, в цей час можна керувати іншими пристроями. За фонове переміщення даних відповідає пристрій DMA (Direct Memory Access).

Контролер має в своєму розпорядженні відразу два контролера DMA, які забезпечують в цілому 16 потоків (по 8 потоків на кожен контролер), кожен з яких призначений для керування запитами до пам’яті від одного або декількох пристроїв. Кожен потік може забезпечувати до 8 каналів (запитів). Кожен контролер DMA має пристрій вирішення конфліктів для обробки запитів відповідно до їх пріоритету.

Контролер DMA дозволяє проводити запис даних в трьох напрямках:

- від периферії в пам’ять;

- з пам’яті до периферії;

- з пам’яті в пам’ять.

Передача ведеться або в режимі безпосередньої передачі, або в режимі

черги. Підтримується різний розмір даних для передачі, причому розмір даних для приймача і джерела може бути неоднаковим. В такому випадку DMA визначає дану ситуацію і виконує необхідні дії для оптимізації передачі, однак ця можливість підтримується лише в режимі черги. Крім того, дуже корисною може виявитися функція циклічної передачі, при використанні якої передача даних починається з початкового адреси знову після передачі останньої одиниці даних джерела, наприклад так можна автоматично оновлювати масив з даними ADC по декількох каналах одночасно.

Програміст може задавати пріоритети для запиту кожного конкретного потоку, інакше пріоритет визначається на основі значень за замовчуваннями.

Для ілюстрації роботи з DMA поставимо задачу циклічної передачі на ЦАП (DAC) даних з масиву, який представляє криву з 16-ма відліками:

```
uint8_t levels[] = {0x00, 0x11, 0x22, 0x33, 0x44,
0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC,
0xDD, 0xEE, 0xFF};
```

В такому виконанні на вихід A4 подаватиметься 1000 раз за секунду поступово зростаюча напруга – пилкоподібний сигнал. Для цього буде зручним винести налаштування в окремі функції:

```
void Init(void);
void InitGpio(void);
void InitTimer(void);
void InitDac(void);
void InitDma(void);
```

які реалізуємо після функції main(). В самій головній функції викликаємо ініціалізацію всіх налаштувань, і залишимо робочий цикл пустим, в який можна додати інші корисні дії. При цьому передача даних до ЦАП буде відбуватися у фоновому режимі і виконання програми не буде перериватися на жоден тактовий імпульс.

```
int main(void)
{
    Init();
    while(1);
}
```

```
void Init(void) {
    InitGpio();
    InitTimer();
    InitDac();
    InitDma();
}
```

Розглянемо послідовно функції ініціалізації.

```
void InitGpio(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);
}
```

Ініціалізація виходу ЦАП схожа на попередні, але тут вихід налаштовано на аналоговий сигнал без підтягування. Інших особливостей по попереднім прикладам тут немає.

DMA вимагає наявності сигналу для передачі наступного числа до регістру ЦАП. Для цього в програмі використано шостий базовий таймер. Від попередніх прикладів ця ініціалізація відрізняється не лише тим що переповнення відбувається 16000 раз за секунду, але й додатково використано функцію TIM_SelectOutputTrigger для закріплення шостого таймеру в якості джерела сигналів початку обміну DMA каналом:

```
void InitTimer(void) {
    TIM_TimeBaseInitTypeDef tim_init;
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM6,
ENABLE);
    tim_init.TIM_CounterMode = TIM_CounterMode_Up;
    tim_init.TIM_Period = SystemCoreClock/1000/160
- 1;
    tim_init.TIM_Prescaler = 10 - 1; //16000Hz
```

```
    TIM_TimeBaseInit(TIM6, &tim_init);
    TIM_SelectOutputTrigger(TIM6,
TIM_TRGOSource_Update);
    TIM_Cmd(TIM6, ENABLE);
}
```

Наступна функція ініціювання ЦАП ще не розібрана, тому на ній потрібно зупинитися більш детально:

```
void InitDac(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_Channel_1, &dac_init);
}
```

Після включення тактування ЦАП, функція DAC_StructInit заповнює структуру значеннями для програмного подання даних. Тому для використання DMA потрібно змінити лише пару параметрів, DAC_Trigger – приймає значення DAC_Trigger_T6_TRGO для визначення використання шостого таймеру в якості сигналу отримання нових даних; DAC_OutputBuffer – вмикається буферизований обмін інформацією записом значення DAC_OutputBuffer_Enable. Після цього приймаються налаштування.

Останньою функцією налаштовується DMA:

```
void InitDma(void) {
```

В першу чергу проводиться увімкнення тактування пристрою DMA.

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1,
                                        ENABLE);
```

Далі оголошується структура налаштувань DMA, після чого це структура переводиться в стан налаштувань по замовчанню. Це дає змогу пропускати поля, які не використовуються в наведеному налаштуванні, бо вони не стосуються використаного режиму роботи.

```
DMA_InitTypeDef dma_init;
```

```
DMA_DeInit(DMA1_Stream5);
```

Наступний параметр визначає канал, який використовуватиметься при передачі масиву в ЦАП.

```
dma_init.DMA_Channel = DMA_Channel_1;
```

Далі потрібно вказати адресу пам'яті куди будуть надходити дані, а також, наступною строчкою, початкову адресу на масив-джерело даних.

```
dma_init.DMA_PeripheralBaseAddr =
                    (uint32_t)&DAC->DHR8R1;
dma_init.DMA_Memory0BaseAddr = (uint32_t)levels;
```

Також умовою правильної роботи є задання напрямку передачі. Тут вказується, що дані передаватимуться від пам'яті до пристрою з використанням буферизації в 16 значень (довжина масиву).

```
dma_init.DMA_DIR = DMA_DIR_MemoryToPeripheral;
dma_init.DMA_BufferSize = 16;
```

Передача даних супроводжується збільшенням адреси приймача та джерела даних, однак пристрій приймає дані за однією адресою. Тому наступні рядки вказують на те, що приймач не буде збільшувати адресу, а джерело – збільшуватиме.

```
dma_init.DMA_PeripheralInc =
                    DMA_PeripheralInc_Disable;
dma_init.DMA_MemoryInc = DMA_MemoryInc_Enable;
```

Також може бути ситуація нерівності розрядності вхідних та вихідних даних. Тому налаштування містять вказівку, що передаються та приймаються байти.

```
dma_init.DMA_PeripheralDataSize =
                    DMA_PeripheralDataSize_Byte;
dma_init.DMA_MemoryDataSize =
                    DMA_PeripheralDataSize_Byte;
```

Режим передачі даних є циклічним, при завершені передачі буферу даних, автоматично передача почнеться з початку.

```
dma_init.DMA_Mode = DMA_Mode_Circular;
```

Робота в прикладі ведеться з одним джерелом та приймачем даних, тому пріоритет можна обрати довільний.

```
dma_init.DMA_Priority = DMA_Priority_High;
```

Наступна пара рядків організує роботу у вигляді прямої передачі даних. В разі організації потоку, при наявних вхідних даних створюється черга. В нашому випадку при відсутності можливості прийому передача буде тимчасово зупинена.

```
dma_init.DMA_FIFOMode = DMA_FIFOMode_Disable;
dma_init.DMA_FIFOThreshold =
                    DMA_FIFOThreshold_HalfFull;
```

Далі визначається пакетування вхідних та вихідних даних. Приклад використовує байтову передачу без пакетування, тому наступна пара налаштувань вказує на використання одного байту.

```
dma_init.DMA_MemoryBurst = DMA_MemoryBurst_Single;
dma_init.DMA_PeripheralBurst =
                    DMA_PeripheralBurst_Single;
```

В останню чергу задіюються налаштування та подаються команди на увімкнення пристроїв та їх початок роботи.

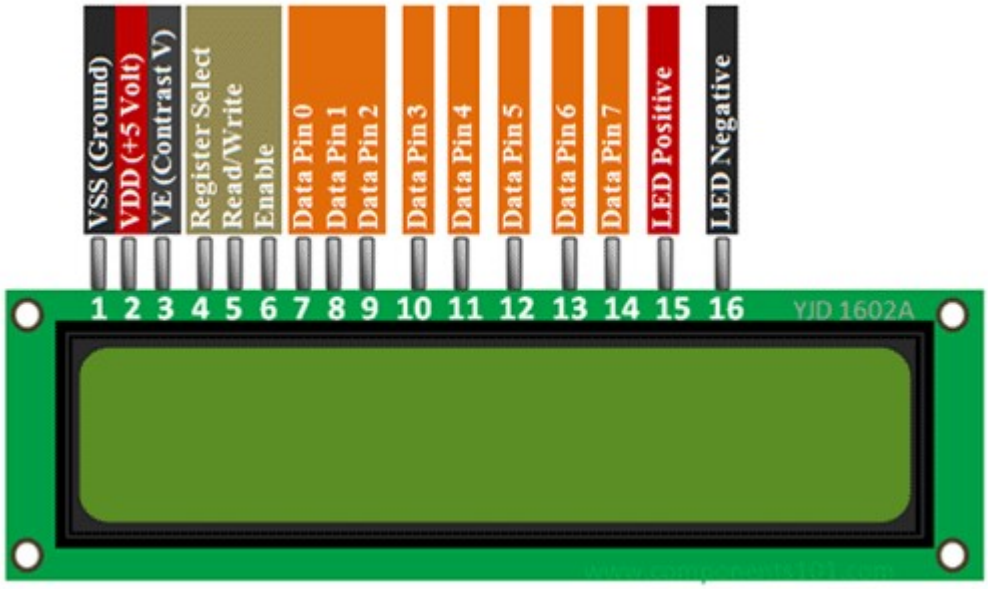
```
DMA_Init(DMA1_Stream5, &dma_init);
DMA_Cmd(DMA1_Stream5, ENABLE);
DAC_Cmd(DAC_Channel_1, ENABLE);
DAC_DMACmd(DAC_Channel_1, ENABLE);
}
```

По цьому програма має в фоновому режимі формувати на виході GPIOA.4 сигнал, графік якого таблично заданий масивом levels.

3.8. Робота з символьним рідкокристалічним екраном

Для зображення інформації для людського ока мікроконтролери не мають спроможності обслуговувати повноцінний монітор за ряду причин, зокрема оперативної пам’яті не вистачить для формування повноцінного кадру. Тому для мікроконтролерних пристроїв використовують дисплеї та індикатори з власною пам’яттю. Ще більш можна заощадити ресурси застосувавши символьні екрани, коли зображення символу задається ASCII кодом. В такому режимі мікроконтролер може ефективно використовувати власні ресурси.

Для текстових повідомлень широко використовують екрани по типу XXX1602X (літерою X позначено кодування різноманітних фірм виробників, рис. 3.14). Такі екрани можуть бути різного розміру, кольорів зображення, але всі вони підпорядковані єдиним протоколом обміну інформації.



3.14. Символьний дворядковий дисплей з позначенням виходів.
https://components101.com/sites/default/files/component_pin/16x2-LCD-Pinout.png

Дисплей використовує три лінії керування та вісім ліній паралельної передачі даних – всього 11 виходів мікросхеми. В деяких випадках, якщо тут можна зменшити швидкодію передачі даних вдвічі, дисплеї використовують в режимі передачі даних по пів-байту, в цьому випадку використовують чотири старших розряди лінії даних. Також, якщо немає потреби читати дані з екрану, можна вихід Read/Write приєднати до «землі» і це звільнить ще один вихід з мікросхеми; в такому форматі мікроконтролер буде використовувати 6 виходів.

Апаратної підтримки використання рідкокристалічного символьного екрану STM мікроконтролери не мають, тому всі дії потрібно виконувати за допомогою системи GPIO.

Instruction	Instruction Code										Description	Execution time (fosc=270Khz)
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear Display	0	0	0	0	0	0	0	0	0	1	Write “00H” to DDRAM and set DDRAM address to “00H” from AC	1.53ms
Return Home	0	0	0	0	0	0	0	0	1	—	Set DDRAM address to “00H” from AC and return cursor to its original position if shifted. The contents of DDRAM are not changed.	1.53ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	SH	Assign cursor moving direction and enable the shift of entire display.	39 μs
Display ON/OFF Control	0	0	0	0	0	0	1	D	C	B	Set display (D), cursor (C), and blinking of cursor (B) on/off control bit.	39 μs
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L	—	—	Set cursor moving and display shift control bit, and the direction, without changing of DDRAM data.	39 μs
Function Set	0	0	0	0	1	DL	N	F	—	—	Set interface data length (DL:8-bit/4-bit), numbers of display line (N:2-line/1-line)and, display font type (F:5×11 dots/5×8 dots)	39 μs
Set CGRAM Address	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0	Set CGRAM address in address counter.	39 μs
Set DDRAM Address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Set DDRAM address in address counter.	39 μs
Read Busy Flag and Address	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Whether during internal operation or not can be known by reading BF. The contents of address counter can also be read.	0 μs
Write Data to RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write data into internal RAM (DDRAM/CGRAM).	43 μs
Read Data from RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read data from internal RAM (DDRAM/CGRAM).	43 μs

＊ ”—” : don’t care

Рис. 3.15. Список доступних команд символьного дисплею

Розглянемо роботу з екраном, а після – приклад програми для передачі

тексту від мікроконтролера. Для цього знадобиться список доступних команд дисплею (рис. 3.15). Також необхідно витримати послідовність ініціалізації дисплею, яку в документації представлено блок-схемою, яку наведено на рис. 3.16. Відповідно блок-схемі, після кожної команди необхідно витримати паузу, яку зазначено в таблиці з рис. 3.15. Акт приймання інформації контролером екрану відбувається в момент подачі на вхід Enable одиниці. При цьому мінімальний час запису інформації (пів-байту) складає 500 нс., тобто 1 мкс. На один байт.

Для роботи мікроконтролера екраном визначимо функціональне призначення виходів за допомогою оголошення define:

```
#define pinD7 GPIO_Pin_9
#define pinD6 GPIO_Pin_11
#define pinD5 GPIO_Pin_13
#define pinD4 GPIO_Pin_15
#define PORT_DATA GPIOE

#define pinE  GPIO_Pin_11
#define pinRS GPIO_Pin_13
#define PORT_CONTROL GPIOB
```

Тут використано різні порти для ліній даних та керування для більш зручного схематичного підключення (менше тягнутися дротами).

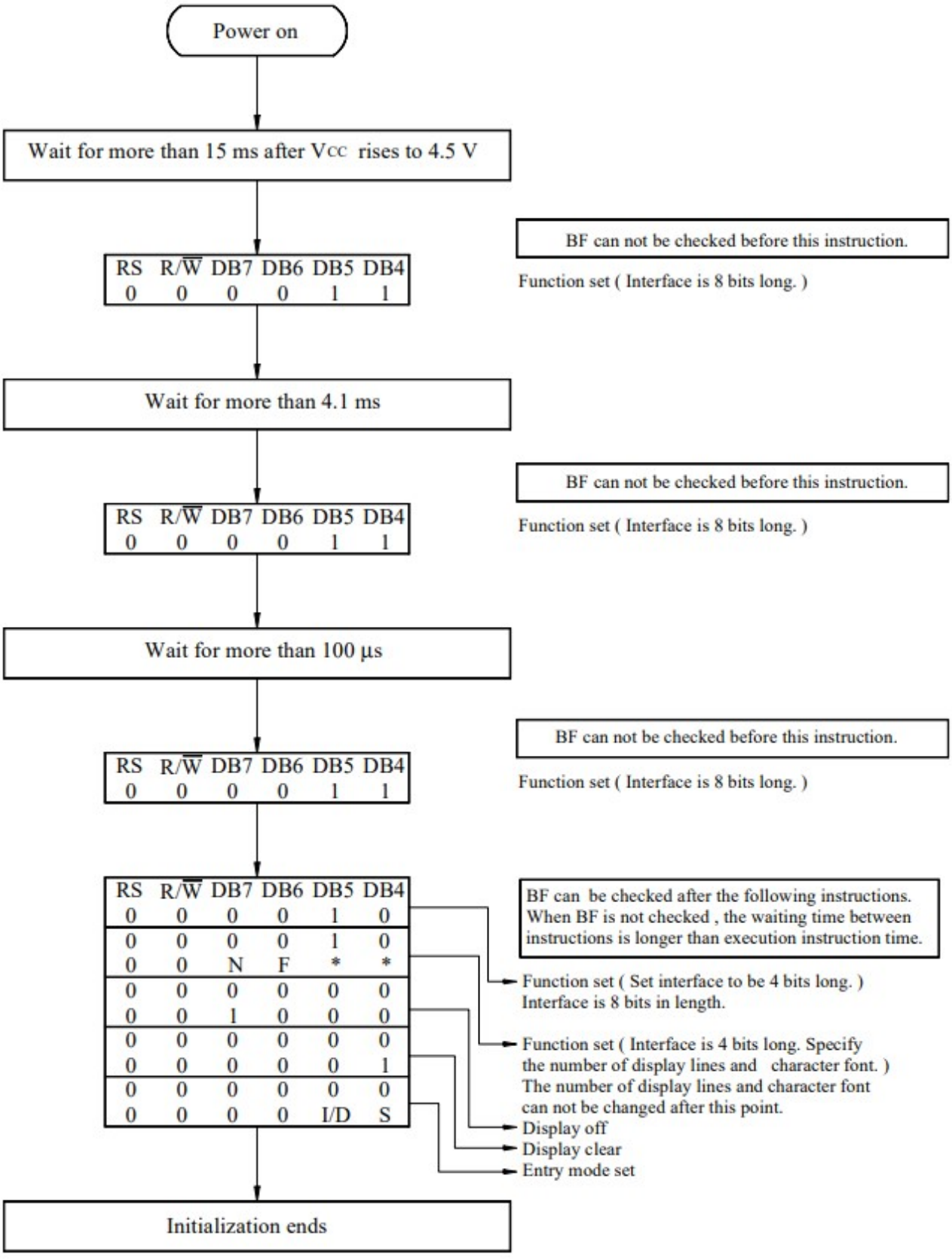


Рис. 3.16. Блок-схема процесу ініціалізації символьного екрану в чотирьохрозрядному режимі передачі даних

Більш зручно використати також макроси й для керування перемиканням

виходів між «0» та «1», це не лише зробить вміст програми більш зрозумілішим та лаконічним, але й дозволить переписати функції на використання інших виходів лише в одному місці програми:

```
#define SET_RS GPIO_SetBits(PORT_CONTROL, pinRS)
#define RESET_RS GPIO_ResetBits(PORT_CONTROL, pinRS)
#define SET_E GPIO_SetBits(PORT_CONTROL, pinE)
#define RESET_E GPIO_ResetBits(PORT_CONTROL, pinE)

#define SET_D7   GPIO_SetBits(PORT_DATA, pinD7)
#define RESET_D7 GPIO_ResetBits(PORT_DATA, pinD7)
#define SET_D6   GPIO_SetBits(PORT_DATA, pinD6)
#define RESET_D6 GPIO_ResetBits(PORT_DATA, pinD6)
#define SET_D5   GPIO_SetBits(PORT_DATA, pinD5)
#define RESET_D5 GPIO_ResetBits(PORT_DATA, pinD5)
#define SET_D4   GPIO_SetBits(PORT_DATA, pinD4)
#define RESET_D4 GPIO_ResetBits(PORT_DATA, pinD4)
```

Для задання пауз використаємо простий пустий цикл:

```
void Pause(volatile int t){
    while(t>0){
        t--;
    }
}
```

Тепер потрібно налаштувати виходи мікросхеми:

```
void lcd_gpio_init(){
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);
    //Після подачі тактування потрібно налаштувати лапки
мікросхеми на вивід інформації
    GPIO_InitTypeDef gi;
    gi.GPIO_Mode = GPIO_Mode_OUT;
    gi.GPIO_OType = GPIO_OType_PP;
    gi.GPIO_Pin = pinD7 | pinD6 | pinD5 | pinD4;
    gi.GPIO_PuPd = GPIO_PuPd_NOPULL;
    gi.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_Init(PORT_DATA, &gi);
    gi.GPIO_Pin = pinE | pinRS;
    GPIO_Init(PORT_CONTROL, &gi);
}
```

Байт до екрану передається парою пів-байтів, цю процедуру описує наступний код:

```
void SetOutputs(char c){
    SET_E; //Встановлюється сигнал на початок передачі
    Pause(200); //Пауза для того, щоб екран встиг
перемкнутися в потрібний режим, після чого виставляємо
біти відповідно старшого пів-байту:
    if( c & 0b10000000 ) SET_D7; else RESET_D7;
    if( c & 0b01000000 ) SET_D6; else RESET_D6;
    if( c & 0b00100000 ) SET_D5; else RESET_D5;
    if( c & 0b00010000 ) SET_D4; else RESET_D4;
    Pause(200); //Пауза за специфікацією екрану
    RESET_E; //Скидаємо лінію синхронізації
    Pause(100); //І в цей час екран захоплює вхідні дані
    SET_E; //Знову переходимо в режим зміни даних і
виставляємо біти молодшого пів-байту
    if( c & 0b00001000 ) SET_D7; else RESET_D7;
    if( c & 0b00000100 ) SET_D6; else RESET_D6;
    if( c & 0b00000010 ) SET_D5; else RESET_D5;
    if( c & 0b00000001 ) SET_D4; else RESET_D4;
    Pause(200); //Пауза для фіксування рівнів
    RESET_E; //При скиданні синхроімпульсу екран захоплює
наступні пів-байту
    Pause(200); //Тут екран «переварює» прийнятий байт
}
```

Операція передачі байту даних та байту команди не відрізняються, що робити екрану з прийнятим байтом вказує напруга по лінії RS. Тому функції передачі команди та даних відрізняються лише зміною стану вказаної лінії:

```
void WriteLcdCommand(char c){
    RESET_RS;
    SetOutputs(c);
}
void WriteLcdData(char c){
    SET_RS;
    SetOutputs(c);
}
```

При наявності таких мінімальних ресурсів можна реалізувати блок-схему ініціювання екрану. Всі дії прокоментовано в тілі функції:

```
void LcdInit(){
    lcd_gpio_init(); //Налаштування виходів
мікроконтролера
    Pause(30000); //Пауза для завантаження
    Pause(30000); //контролеру екрану
    RESET_RS; //Далі режим запису команд
```

```
    SET_E; //По блок-схемі з рис. 3.16 записуємо пів-
байти
    RESET_D7;
    RESET_D6;
    SET_D5;
    RESET_D4;
    RESET_E;
    Pause(200);
    SET_E;
    RESET_D7;
    RESET_D6;
    SET_D5;
    RESET_D4;
    RESET_E;
    Pause(200);
    SET_E;
    SET_D7;
    SET_D6;
    RESET_D5;
    RESET_D4;
    RESET_E;
    Pause(200);
    Pause(8000);
    WriteLcdCommand(0b00001111); //Вмикаємо дисплей (див.
табл. 3 рис. 3.15)
    Pause(8000);
    WriteLcdCommand(0b00000001); //Чистимо дисплей
    Pause(32000);
    WriteLcdCommand(0b00000110); //При записі байту
курсор рухатиметься на символ праворуч
    Pause(8000);
    WriteLcdCommand(0x0E); //Вмикаємо дисплей та робимо
видимим курсор
    Pause(8000);
    WriteLcdCommand(0x01); //Повторне очищення дисплею
    Pause(16400);
}
```

Встановлювати курсор на екрані можна зо допомогою команди 0b1xxxxxxx, де іксами позначено двійкове число-адреса «відеопам'яті» – перший рядом має адреси від 0x00 до 0x0F, а другий рядок має адреси від 0x40 до 0x4F. Відповідно до цієї інформації можна побудувати функцію встановлення курсору в потрібну позицію:

```
void ToXY(int y, int x){//y=0,1; x=0,1,..,15
```

```
    WriteLcdCommand(0x80 | (y<<6) | x);
}
```

На цьому етапі є всі засоби для використання екрану для виводу інформації:

```
int main(void)
{
    int p=0;
    SystemInit();
    SystemCoreClockUpdate();
    LcdInit(); //Застосування налаштувань для роботи
екрану
    Pause(32000);
    WriteLcdCommand(0x80); //Команда початку запису даних
на адресу початку першого рядка
    WriteLcdData('H'); //Дані передаються як символи
    WriteLcdData('e');
    WriteLcdData('l');
    WriteLcdData('l');
    WriteLcdData('o');
    WriteLcdData('!');
    WriteLcdCommand(0xC0); //Команда початку запису даних
на адресу початку другого рядка
    WriteLcdData('W');
    WriteLcdData('o');
    WriteLcdData('r');
    WriteLcdData('l');
    WriteLcdData('d');
    WriteLcdData('!');
// Створення власного символу
```

Символьні екрани також підтримують використання власних символів з кодом від 0 до 7 включно. Для цієї мети з адресу 0x40 знаходяться бітові зображення користувацьких символів. Наприклад потрібно додати до таблиці символів знак смайлику, тоді необхідно мати наступний масив з бітовим зображенням:

```
char my_char[]={
    0b00001110,
    0b00010001,
    0b00011011,
    0b00010001,
    0b00011111,
    0b00010001,
```

```
    0b00001110,
    0b00000000
};
```

Ссимвольний генератор для побудови зображення використовує лише п’ять молодших бітів, формувавши таким чином шрифт 5x8 точок. Щоб занести зображення символу до екрану, потрібно байти бітового зображення відправити починаючі з адреси 0x40 + 8*code, де code – номер власного символу від 0 до 7:

```
    char code = 0;
    WriteLcdCommand(0x40 + 8*code);
    for(p=0;p<8;p++){
        WriteLcdData(my_char[p]);
    }
```

Далі виведемо цей символ в першому рядку на шостому знакомісті:

```
    WriteLcdCommand(0x80 + 6);
    WriteLcdData(code);

    while(1)
    {
    }
}
```

Далі в програмі йде пустий цикл, бо зміни тексту на екрані не передбачено.

Але програма є не повною, бо в ній відсутні можливості виводити строки, відображати числа в десятковому форматі, неможливо виконувати вивід інформації у фоновому режимі (за допомогою переривання від таймеру). Також використавши таблицю з рис. 3.15 можна реалізувати функцію встановлення курсору в позицію, додавання власного символу, очищення екрану та інші.

Плюсом такого підходу є те, що швидкість виводу тексту на екран обмежена можливостями самого екрану, але нижня швидкість та рівномірність подачі інформації не регламентовано. Завдяки цьому текст до екрану можна передавати вкрай не рівномірно та іноді з досить великими паузами, тому цей процес досить легко поєднати з виконанням більш важливих за таймінгом процесами.

Бібліографія

1. Jack Ganssle and Michael Barr. 2003. Embedded Systems Dictionary. CMP Books.

2. Уилмсхерст Т. Разработка встроенных систем с помощью микроконтроллеров PIC. –М.: МК-Пресс. 2008, 544с

3. Arnold S. Berger. Embedded Systems Design: An Introduction to Processes, Tools, and Techniques. CMP Books © 2002. ISBN: 1-57820-073-3

4. Роберт Гласс. Факты и заблуждения профессионального программирования. "Символ", 2008. ISBN 13: 978-5-93286-092-2

5. Конспект лекцій з дисципліни «Програмування систем реального часу» напрям підготовки 6.050202 «Автоматизація та комп’ютерно-інтегровані технології» / Укладачі : Чихіра І.В., Микитишин А.Г., – Тернопіль : ерноп ільський національний технічний університет імені Івана Пулюя , 2016. – 76 с.

6. Datasheet stm32f407/ офіційна документація / електронний ресурс: https://www.st.com/resource/en/datasheet/dm00037051.pdf