Міністерство освіти і науки України
Міжнародний гуманітарний університет
Кафедра комп’ютерних наук

Русу О.П.

ПРОГРАМУВАННЯ МІКРОКОНТРОЛЕРІВ ТА РОБОТОТЕХНІКА В ЗАКЛАДАХ ОСВІТИ

Конспект лекцій

Одеса 2025

Затверджено Вченою Радою Міжнародного гуманітарного університету.
Протокол № 5 від 20 січня 2025 р.

Русу О.П. Програмування мікроконтролерів та робототехніка в закладах освіти. Конспект лекцій [Електронне видання]. Кафедра комп'ютерних наук Міжнародного гуманітарного університету. Одеса, 2025. 129 с.

Дисципліна «Програмування мікроконтролерів та робототехніка в закладах освіти» присвячена вивченню принципів побудови та функціонування мікроконтролерів, а також здобуттю навичок їх практичного використання. Особливу увагу у курсі приділяється створенню програмного забезпечення на мовах асемблеру та принципам побудови пристроїв Інтернету речей. Розглядаються питання викладання програмування мікроконтролерів та робототехніка в закладах освіти.

Автор висловлює подяку Івану Панкову за активну роботу у підготовці цих матеріалів.

ЗМІСТ

ВВЕДЕННЯ В ДИСЦИПЛІНУ 4
ПРОГРАМА «ВІТАЮ, СВІТЕ!» 17
ПІДПРОГРАМИ ТА ЗАТРИМКИ 37
ФОРМУВАННЯ ЧАСОВИХ ІНТЕРВАЛІВ 52
ТАЙМЕРИ ТА ПЕРЕРИВАННЯ 72
УМОВНІ ОПЕРАЦІЇ 90
ПРОГРАМУВАННЯ ПРИСТРОЇВ ІНТЕРНЕТУ РЕЧЕЙ.....108

ВВЕДЕННЯ В ДИСЦИПЛІНУ

Сьогодні ми починаємо вивчати нову дисципліну, яка називається «Програмування мікроконтролерів та пристроїв інтернет речей». Питання, пов’язані безпосередньо з Інтернетом речей, ми розглянемо наприкінці курсу, тому що більшість пристроїв, що відноситься до цієї категорії – це застосунки на мікроконтролерах, з певними специфічними особливостями. Тому основну частину курсу ми присвятимо безпосередньо вивченню мікроконтролерів та особливостей їх програмування.

Насамперед варто з’ясувати, що таке мікроконтролер. Мікроконтролер – це комп’ютер, реалізований у вигляді інтегральної мікросхеми. Він подібний до мікропроцесора, який також є мікросхемою. Але мікроконтролер має одну принципову відмінність – мікропроцесору необхідні для роботи зовнішній тактовий генератор, оперативна та постійна пам’ять, інтерфейси для зв’язку з іншими компонентами системної плати, а також певна кількість додаткових вузлів. А в мікроконтролері ці елементи вже інтегровані. Тобто мікроконтролер є самостійним пристроєм, який не потребує додаткових зовнішніх вузлів для роботи.

Типовий мікроконтролер містить: ядро, енергонезалежну пам’ять програм, оперативну пам’ять і щонайменше один генератор тактових сигналів. Але головною відмінністю мікроконтролера від мікропроцесора є наявність різноманітних вбудованих периферійних модулів, які на жаргоні розробників електроніки називаються просто «периферією». Під периферійними вузлами розуміють апаратні вузли, призначені для виконання конкретних функцій, які безпосередньо підтримують роботу з різними об’єктами чи інтерфейсами. Саме наявність таких модулів в мікроконтролері і визначає його основне призначення – керування об’єктами або системами.

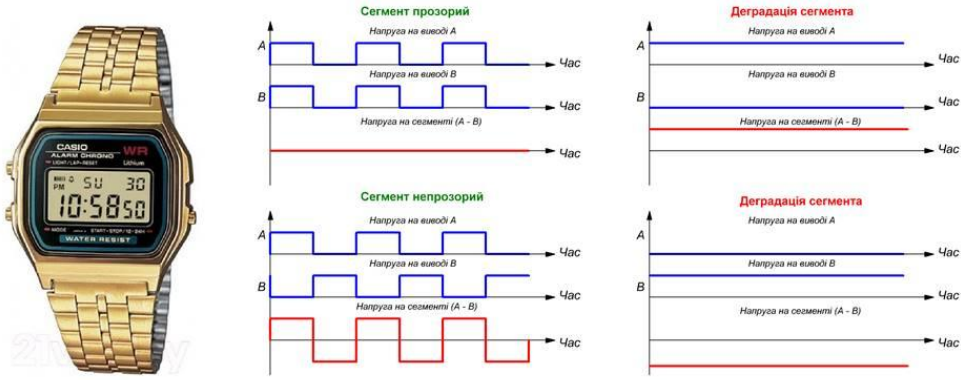
Мікроконтролер

Мікроконтролер (Microcontroller) – однокристальний мікрокомп'ютер, виконаний у вигляді мікросхеми

Склад мікроконтролера
- Мікропроцесор
- Пам’ять програм (енергонезалежна)
- Оперативна пам’ять (енергозалежна)
- Генератори тактових сигналів
- Периферійні пристрої

Прикладом специфічного периферійного модуля мікроконтролера є драйвер рідкокристалічного індикатора. Особливістю рідкокристалічних індикаторів є необхідність подачі на електричні виводи його сегментів напруги змінної полярності з частотою близько 100 Гц – інакше сегменти будуть деградувати і індикатор перестане працювати. Таку напругу можна сформувати програмним шляхом – ця задача не є складною. Однак вирішення її програмним шляхом потребує певної кількості процесорного часу – ресурсу, якого завжди не вистачає. Але якщо в мікроконтролері є спеціалізований периферійний модуль – драйвер рідкокристалічного індикатора, то програмісту потрібно лише один раз його налаштувати, а потім тільки своєчасно оновлювати інформацію, яку необхідно відображати на екрані. Усі необхідні електричні сигнали у цьому випадку тепер будуть формуватися апаратно без участі процесорного ядра.

Принцип керування рідкокристалічним індикатором



Типовими периферійними модулями мікроконтролера є порти введення-виведення загального призначення – вони є в кожному мікроконтролері, оскільки через них мікроконтролер з'єднується з іншими частинами електричної схеми. Крім того, до складу периферійних модулів входять різноманітні таймери, за допомогою яких можна формувати сигнали в певної тривалості та функціонувати у реальному часі. Для роботи з аналоговими сигнали використовують операційні підсилювачі, компаратори, аналого-цифрові та цифро-аналогові перетворювачі. Крім цього, мікроконтролери можуть містити готові приймально-передавальні вузли для поширених інтерфейсів передачі даних, серед яких UART, SPI, I^2C, CAN, 1-Wire, IrDA, USB, Bluetooth, Wi-Fi та багато інших. Усе це свідчить, що мікроконтролер орієнтований насамперед на керування фізичними об'єктами та технічними вузлами на відміну мікропроцесора, який може виконувати лише арифметико-логічні операції. Власне, саме від цього походить його назва: «controller» у перекладі з англійської означає «керувати».

Мікроконтролер STM8S003F3

- 16-розрядний таймер з розширеними функціями (4 канали порівняння, 3 канали з комплементарними виходами, таймер «мертвого» часу, підтримка зовнішньої синхронизації)
- 16-розрядний таймер загального призначення (3 канали порівняння)
- 8-розрядний таймер загального призначення
- Таймер автоматичного пробудження
- Сторожовий таймер
- Модуль послідовних інтерфейсів (UART, SmartCard, IrDA, LIN)
- Модуль інтерфейсу SPI (до 8 Мбіт/с)
- Модуль інтерфейсу I^2C (до 400 кбіт/с)
- 10-розрядний 5-канальний аналого-цифровий перетворювач
- До 28 каналів портів введення-виведення загального призначення

Варто звернути увагу на правильний наголос у слові «мікроконтрОлер». Часто можна почути помилкове наголошування –«мікроконтролЕр», що фактично змінює зміст слова. «МікроконтролЕр» – це «маленька людина, що перевіряє правильність побудови будиночка для ляльок», тоді як у нашій дисципліні йдеться саме про мікросхему, призначену для керування електронними пристроями. Тому правильний наголос у цьому слові буде саме на другу літеру «О» – «МікроконтрОлер».

Правильний наголос

МікрокотролЕр **МікроконтрОлер**

Практична частина нашого курсу розрахована на використання мікроконтролера PIC16F84, тому з ним варто познайомитися більш детально. Оригінальна версія цієї мікросхеми вже не випускається, натомість існує і доступна її більш досконала модифікація з суфіксом «А». Різницю між цими двома мікросхемами знає лише виробник, ми ж надалі будемо вважати, що мікросхеми PIC16F84 та PIC16F84A повністю тотожні.

Мікроконтролер PIC16F84 належить до середнього сімейства 8-розрядних мікроконтролерів компанії Microchip Technology. Його тактова частота може досягати 20 МГц, хоча більшість мікросхем, що є на ринку, розрахована на роботу з тактовими частотами до 4 МГц. Обсяг енергонезалежної пам’яті програм (Program Memory), реалізованої за технологією Flash, дорівнює 1024 командам. Оперативна пам’ять (Random Access Memory, RAM) має дуже скромний обсяг – лише 68 байт, але виявляється, що цього цілком достатньо для багатьох застосунків. Крім того цей мікроконтролер має 64 байтb енергонезалежної пам’яті з електричним стиранням (Electrically Erasable Programmable Read-Only Memory EEPROM), яку можна використовувати, наприклад, використовується для збереження налаштувань.

Набір периферійних модулів у PIC16F84 мінімальний – один восьмирозрядний таймер з мінімальною кількістю функціональних можливостей і 13 каналів портів введення-виведення, загального призначення. Діапазон робочої напруги становить від 2 до 5,5 В, а випускається мікросхема в трьох типах корпусів.

Цікавою є і цінова характеристика. Вартість PIC16F84 становила близько 6 доларів США, що для настільки простого мікроконтролера є досить високою ціною. Нині за таку ж вартість можна придбати значно продуктивніші моделі з розширеними можливостями та більшим набором периферії. Однак ці мікроконтролери ще випускаються і продаються.

PIC16F84A

DIP18

SOIC20

- Розрядність – 8 біт
- Тактова частота – до 20 МГц
- Пам’ять програм – 1024 інструкцій
- Оперативна пам’ять (RAM) – 68 байт
- Енергонезалежна пам’ять (EEPROM) – 64 байт
- Кількість портів вводу-виводу – 13
- Периферійні вузли: 8-розрядний таймер
- Додаткові функції: сторожовий таймер, таймер затримки виконання програми, 4 типи тактових генераторів, 4 джерела переривань
- Діапазон робочої напруги – 2…5,5 В
- Корпус: DIP18, SOIC-18, SOIC-20
- **Вартість ≈ 6USD**

Причина, чому для навчання було обрано саме мікроконтролер PIC16F84, полягає у його історичному значенні. Він з’явився у 1990-х роках і став справжнім проривом у галузі

електроніки. На той час цей мікроконтролер, порівняно з іншими моделями, виявився простим, дешевим та простим в освоєнні, що зробило його надзвичайно популярним як серед аматорів, які нарешті отримали можливість створювати застосунки на мікроконтролерах без необхідності залучення дорогого обладнання, так і серед професійних інженерів і розробників електроніки.

З того часу для PIC16F84 було створено велику кількість програмного коду, включно з цікавими прикладами застосувань. Попри появу сучасніших моделей, цей мікроконтролер і досі випускається компанією Microchip. Причому існують нові і більш дешеві мікроконтролери, які можна встановлювати замість PIC16F84 навіть без корекції програмного забезпечення. За словами представників компанії, ця модель стала легендою і певним брендом, від якого важко відмовитися. Саме тому її й досі використовують для навчання.

Структурна схема мікроконтролера досить проста: арифметико-логічний пристрій, оперативна пам’ять і кілька периферійних модулів. Незважаючи на невисоку продуктивність за сучасними мірками, він придатний для застосування у простих радіоаматорських конструкціях – від ялинкових гірлянд до елементів автоматики. Однак його головна цінність сьогодні полягає в освітньому аспекті: PIC16F84 ідеально підходить для знайомства з принципами роботи мікроконтролерів.

При роботі з мікроконтролерами треба звертати увагу на декілька речей. У першу чергу необхідно мати базові знання з електроніки, зокрема розуміти, як будуються найпростіші електронні схеми та як підключати мікроконтролер до різних вузлів. Також потрібно знати основи програмування, адже для роботи з мікроконтролером

використовуються ті ж самі алгоритмічні конструкції: цикли, безумовні та умовні переходи та підпрограми.

Проте у реальному проєкті недостатньо лише написати програмне забезпечення – його потрібно ще перевірити та відлаштувати. А оскільки мікроконтролер працює тільки у режимі реального часу і керує реальними електронними пристроями, то дуже часто для перевірки працездатності програмного забезпечення доводиться користуватися різноманітними вимірювальними приладами: мультиметрами, логічними аналізаторами, осцилографами тощо.

Крім того, треба знати особливості проєктування друкованих плат, на яких будуть працювати мікроконтролери, бо бувають випадки коли схема і програмне забезпечення спроєктовані правильно, але через невдале розташування компонентів на друкованій платі пристрій працює нестабільно або взагалі неправильно. Іноді дрібні зміни, наприклад переміщення компонента на кілька міліметрів, подовження з'єднувальних провідників або зміна їх геометрії, можуть стати причиною появи непередбаченої завади, яка призведе до нестабільної роботи мікроконтролера.

І останній важливий момент, про який не можна забувати при роботі з мікроконтролерами – розуміння процесу виробництва електроніки. Слід розуміти, що програмне забезпечення пишеться один раз, а електронні компоненти встановлюють в кожен пристрій. Тому якщо є хоча б найменша можливість зменшення кількості електронних компонентів, навіть за рахунок істотного ускладнення програмного забезпечення, то інвестор завжди обере останній варіант – один раз написати складне програмне забезпечення, ніж потім витрачати кошти на встановлення електронних компонентів, без яких можна було б обійтись.

Мікроконтролери сьогодні застосовуються практично у всіх сферах електроніки. Вони зустрічаються у дитячих іграшках, побутовій техніці, промисловому обладнанні, засобах автоматизації, телекомунікаційних системах, дронах, роботах, квадрокоптерах, системах «розумний будинок» та пристроях Інтернету речей. Кондиціонери, ліфти та монітори керуються мікроконтролерами. Автомобільна та авіаційна техніка, космічні кораблі, супутники – всюди можна знайти мікроконтролери, що виконують контролювальні та керувальні функції. Сьогодні простіше знайти застосунки, де мікроконтролери не використовуються, ніж перелічувати сфери їх практичного використання.

Застосування мікроконтролерів

Квадрокоптер

Моноколесо

Смартфон

Робот

- Промислова техніка
- Побутова та офісна техніка
- Засоби автоматизації
- Телекомунікаційні системи
- Автомобілі
- Електротранспорт
- Дрони, квадрокоптери
- Роботи
- Системи «Розумний будинок»
- Пристрої Інтернету речей

Мікроконтролери виробляють достатньо велика кількість компаній, що займаються розробкою електронних компонентів. Кожна компанія пропонує власні рішення, оптимізовані для певних сфер застосування. На ринку постійно відбуваються зміни: одні компанії зникають, інші з'являються або об'єднуються. Наприклад, компанія Atmel свого часу конкурувала з Microchip, але згодом Microchip придбала Atmel, і конкуренція між ними зникла. Це відбулося через появу компанії STMicroelectronics, яка запропонувала дешевші та зручніші мікроконтролери, що змінило розстановку сил на ринку.

Компанія Texas Instruments займає особливе місце в галузі електроніки, виробляючи широкий спектр компонентів, включно з мікроконтролерами. Вона розробляє рішення загального застосування та спеціалізовані лінійки, наприклад для автомобільної промисловості, де застосовуються мікроконтролери підвищеної надійності, що контролюють роботу систем освітлення, гальм та подушок безпеки.

Виробники мікроконтролерів

- Altera
- Analog Devices
- ARM
- **Atmel**
- Cypress Semiconductor
- ELAN Microelectronics Corp.
- EPSON Semiconductor
- Espressif Systems
- Freescale Semiconductor
- Fujitsu
- Holtek
- Hyperstone
- Infineon
- Intel
- Lattice Semiconductor
- Maxim Integrated
- **Microchip Technology**
- National Semiconductor
- NEC
- Nordic Semiconductor
- NXP Semiconductors
- Nuvoton Technology
- Panasonic
- Parallax
- Rabbit Semiconductor
- Raspberry Pi Foundation
- Renesas Electronics
- Redpine Signals
- Rockwell
- Silicon Laboratories
- Silicon Motion
- Sony
- Spansion
- **STMicroelectronics**
- Synopsys
- **Texas Instruments**
- Toshiba
- Ubicom
- WCH
- Western Design Center
- Xemics
- Xilinx
- XMOS
- ZiLOG

Якщо ви будете працювати з мікроконтролерами на промисловому рівні, то ви обов'язково познайомитесь з професійними рішеннями. А зараз – коли ви знаходитесь на самому початковому рівні – вам потрібно лише познайомитися із найпоширенішими лінійками мікроконтролерів загального призначення, які можна придбати у будь-якому магазині, що торгує електронними компонентами.

До найпоширеніших лінійок, що пропонує компанія Microchip відносяться лінійки PIC10, PIC16, PIC24 і PIC32. Мікроконтролер PIC16F84 належить до лінійки PIC16, яку компанія Microchip позиціоную як 8-розрядні мікроконтролери середнього сімейства. Лінійки 32-розрядних мікроконтролерів цієї компанії особливого поширення не набули.

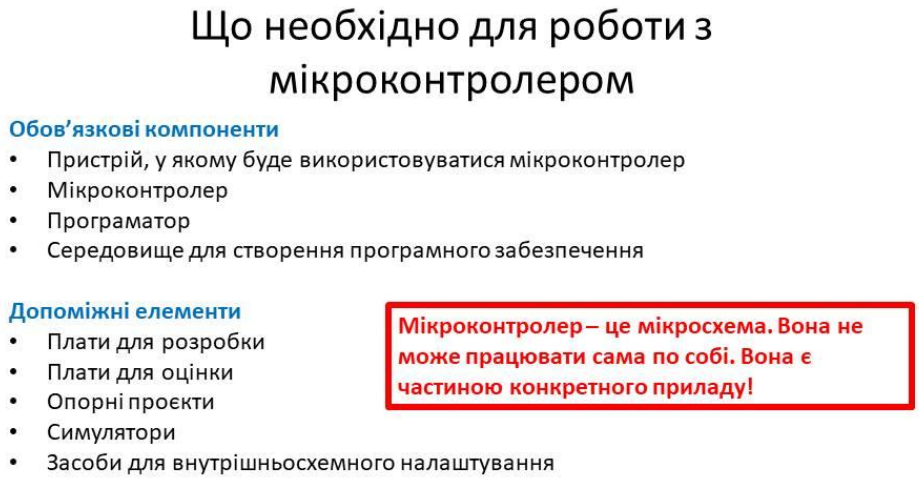
Ще нещодавно основним конкурентом мікроконтролерам PIC були мікросхеми AVR, створені компанією Atmel. Однак після того, як компанія Microchip придбала компанію Atmel подальший розвиток лінійок AVR майже припинився.

Зверніть увагу на компанію STMicroelectronics, мікроконтролери STM8 та STM32 якої стають все більш популярними з кожними роком. Ці недорогі мікросхеми мають надзвичайно високе співвідношення ціна/продуктивність, що робить їх дуже привабливими для все більш широкого кола як професійних розробників, так і аматорів-початківців.

Вибір моделі мікроконтролера залежить або від конкретного технічного завдання, де може вказуватись конкретна мікросхема або лінійка, або від досвіду розробника. Зазвичай розробники електроніки надають перевагу тим мікроконтролерам, з якими вони вже колись працювали і мають позитивний досвід їх використання. Тому виробники мікроконтролерів забезпечують усебічну підтримку розробників, надаючи їм різноманітні інструменти, що дозволять скоротити час на вирішення конкретної задачі.

7,5

От про ці інструменти ми і поговоримо у другій частині лекції. А поки що давайте з'ясуємо, а що ж нам потрібно для того, щоб створити застосунок на основі мікроконтролера. Програмування мікроконтролерів істотно відрізняється від програмування на звичайному комп'ютері. На комп'ютері програму можна написати та одразу перевірити. У випадку мікроконтролера все складніше, оскільки він призначений для керування конкретними електронними схемами. Тому, перше, що потрібно нам для перевірки працездатності програмного забезпечення на мікроконтролері – це електронна схема, якою буде керувати мікроконтролер.



Щоб програму, яку ви створили, записати в мікроконтролер, потрібен програматор – спеціальний пристрій, який завантажує код у мікросхему по спеціальному інтерфейсу. А

саме програмне забезпечення створюється у спеціальному середовищі для створення програмного забезпечення, яке називається інтегроване середовище для розробки (Integrated Development Environment, IDE).

Отже, у загальному випадку програміст, що працює з мікроконтролерами, основну частину часу пише код в спеціалізованому програмному забезпеченні. Потім, за допомогою програматора завантажує скомпільований код в мікроконтролер, далі підключає мікроконтролер до електронної схеми і перевіряє роботи. Якщо усе працює, то можна рухатись далі, а якщо ні – то треба шукати помилку.

Перші мікроконтролери, серед яких і PIC16F84, мали доволі специфічні інтерфейси програмування, що потребувало фізичного відключення мікроконтролера від схеми та підключення до програматора. Тому мікроконтролер PIC16F84 для завантаження програми вставляється у спеціальну панельку в програматорі. Потім мікросхему потрібно було виймати з програматора та вставляти у таку ж саму панельну на схемі.

Це було дуже незручно, особливо при серійному виробництві. Тому з часом майже усі виробники мікроконтролерів опанували технології внутрішньосхемного програмування, яке дозволяло завантажувати та відлагоджувати програмне забезпечення без фізичного відключення мікроконтролера від електричної схеми, в якій він встановлений. Особливо актуально це для мікроконтролерів в компактних корпусах, які інколи просто фізично не можна вставити в програматор.

Сьогодні у більшості пристроїв, що використовують мікроконтролери, на платі передбачена можливість підключення програматора. Це можуть бути різноманітні роз’єми,

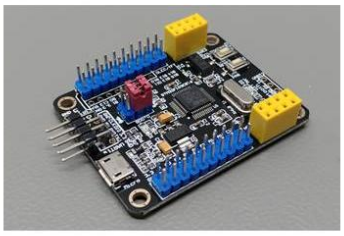
контакти або ще якійсь інші засоби, за допомогою яких можна оновити програмне забезпечення мікроконтролера без фізичного відключення його від схеми.

Тепер розглянемо інше питання. Припустимо у вас виникло необорне бажання стати програмістом мікроконтролерів. Але ви ніколи раніше не тримали паяльника в руках і не збирали електронні схеми. Якщо ви спробуєте створити проєкт «з нуля», тобто придбати електронні компоненти, з'єднати їх між собою, підключити програматор та завантажити програму, то скоріш за все у вас нічного не вийде.

Мікроконтролер сам по собі не працює: окрім вбудованих вузлів, йому потрібна мінімальна зовнішня обв'язка. Обов'язково потрібні блокувальні конденсатори, хоча б один по живленню, а також кварцевий резонатор, без якого мікроконтролер не запуститься (наприклад PIC16F84, де немає вбудованого RC-генератора, тому для його тактового генератора обов'язково потрібні зовнішні електронні компоненти). Будь-яка неточність при зборці електричної схеми або неякісна пайка зовнішніх електронних компонентів призведуть до того, що мікроконтролер не буде подавати «ознак життя» – навіть програма туди не буде завантажуватися. І ви, у такому разі, можете надовго загрузнути у пошуках апаратних проблем, пов'язаних, у більшості випадків, із банальним «людським фактором».

Щоб швидше подолати цей етап невизначеності, виробники мікроконтролерів пропонують апаратні засоби – плати для розробки (Development Board). На таких платах мікроконтролер вже встановлений разом із мінімальною необхідними для його роботи електронним компонентами («обв'язкою»). Навіть на самій простій платі для розробки є роз'єм для підключення програматора, що дозволяє одразу завантажити програму і перевірити працездатність мікросхеми. Тому використання плат для розробки є класичним підходом при створенні дослідних зразків – пристроїв, що дозволяють перевірити працездатність первинної ідеї, що є першим етапом розробки будь-якого апаратного застосунку на основі мікроконтролерів.

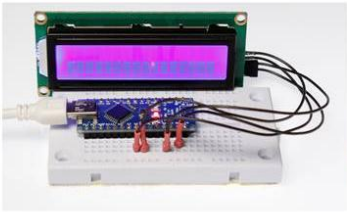
Плата для розробки



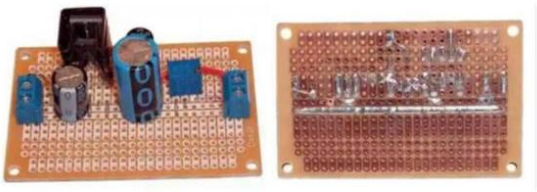
Плата для розробки (Development Board) – друкована плата з уже встановленим мікроконтролером та мінімально необхідними для роботи компонентами. До іншої схеми та програматора плата підключається за допомогою спеціальних роз'ємів (зазвичай із кроком 2,54 мм). **Дозволяє швидко почати роботу із мікроконтролером**

Однак щоб швидко зібрати електричну схему однієї плати для розробки недостатньо, тому що вона містить лише компоненти, необхідні для роботи мікроконтролера. Для швидкого створення електричних схем зараз активно використовуються спеціальні засоби, які називаються макетними платами. Сьогодні існують два види макетних плат: для пайкового та безпайкового з'єднання компонентів. Кожен розробник обирає той вид макетування, який йому більше до вподоби. Початківці зазвичай використовують безпайкові макетні плати, які дозволяють зібрати електричну схему за лічені хвилини. Більш досвідчені розробники надають перевагу з'єднанню за допомогою пайки, яка забезпечує більшу надійність електричних з'єднань, що особливо важливо у великих проєктах, що потребують значних часових витрат.

Макетна плата



Безпайкова макетна плата



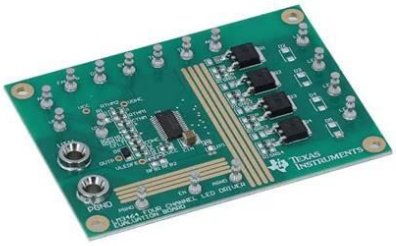
Макетна плата для монтажу методом пайки

Макетна плата (Breadboard) — універсальна плата для складання і моделювання прототипів електронних пристроїв. **Використання макетних плат дозволяє швидко створити схему для того, щоб перевірити її працездатність**

На плати для розробки дуже схожі плати для оцінки (Evaluation Board). На відміну від плат для розробки, призначених для пришвидшення створення кінцевого застосунку, плати для оцінки призначені для всебічного дослідження конкретного електронного компонента. У якості дослідного компонента, можу бути транзистор, мікроконтролер, збірка, перетворювач інтерфейсів або будь яка інша спеціалізована мікросхема.

Особливістю плат для оцінки є наявність допоміжних компонентів, що дозволяють швидко її конфігурувати для роботи в різних режимах та підключати вимірювальні прилади. Плати для оцінки спрощують перевірку роботи компонентів і усувають ризики помилок фізичного макетування, таких як неправильне паяння або некоректне розведення доріжок на друкованій платі. Вони дозволяють побачити, як компонент працює в найкращих – з точки зору виробника – умовах. Разом із платами для розробки, плати для оцінки широко застосовуються у стартапах і прототипуванні для швидкої перевірки технічних рішень.

Плата для оцінки



Плата для оцінки (Evaluation Board) – призначена для оцінки можливостей певного електронного компонента. Містить готову для використання електронну схеми (часто із можливістю зміни конфігурації), а також роз'єми для підключення контрольно-вимірювальної апаратури. **Плати для оцінки дозволяють швидко додати у прилад готовий вузол**

Остання категорія плат, про які сьогодні також потрібно обов'язково згадати, називається «опорні проекти» (Reference Design). Опорний проєкт – це повністю готовий пристрій, який надає виробник розробнику електронного обладнання. В опорному проєкті є друкована плата з мікроконтролером, на платі встановлені усі необхідні електронні компоненти, а в мікроконтролер завантажено програмне забезпечення з відкритим вихідним кодом.

Виробник зазвичай забороняє використовувати такі плати для серійного виробництва, але для прототипування та тестування їх можна використовувати без обмежень. Опорні проекти дозволяють розробляти власні пристрої на базі готових рішень, не починаючи «з нуля». Наприклад, можна взяти готовий інгалятор і модифікувати його для спеціального приміщення, змінюючи форсунку, матеріали або програмне забезпечення для досягнення необхідного результату.

Використання опорних проєктів дозволяє швидко створювати MVP-проекти (Minimum Viable Product – мінімально життєздатні продукти) з мінімальними витратами часу

та ресурсів. Опорні проекти значно прискорюють розробку нових пристроїв і дають можливість зосередитися на унікальних функціях, не витрачаючи час на базову реалізацію та інтеграцію мікроконтролера.

Я певен, ви вже зрозуміли, що сучасні електронні пристрої фактично складаються із стандартних вузлів, які можна придбати на готових платах та з’єднати у єдину систему впродовж кількох хвилин не чіпаючи паяльник. Це недешево, але швидко – а в наш час швидкість – це один із головних показників будь-якого проєкту.

Тепер поговоримо про програматори. Програматори – це пристрої, призначені для запису інформації в мікросхеми. Вони можуть працювати не лише з мікроконтролерами, а й з мікросхемами пам’яті, програмованими логічними інтегральними схемами та іншими компонентами. До програматора зазвичай додається спеціальна програма з драйвером, яка формує сигнали, зрозумілі мікросхемі, куди відбувається запис.

Основна функція перших програматорів полягала у запису інформації в мікросхему та/або читання інформації з неї. До цих функцій додавалася функція перевірки правильності запису, яка полягала у зчитуванні записаної і мікросхему інформації з подальшою перевіркою.

Однак перелік функцій сучасних програматорів значно розширився. Тепер програматори підтримують внутрішньосхемне програмування та відлагодження програми, що дозволяє зупинити роботу мікроконтролера та визначити його внутрішній стан у певній точці (точці зупинки) програмного забезпечення. Крім того, однією із стандартних задач, яку треба реалізувати в сучасних застосунках, є обмін інформацією між мікроконтролером та

комп'ютером. Тому сучасні програматори можуть мати один або декілька інформаційних каналів, кожен з яких містить перетворювачі простих інтерфейсів, що використовуються в мікроконтролерних системах (UART, SPI, I^2C, тощо), в складні інтерфейси, що використовуються в комп'ютерах (наприклад, USB).

Основні функції програматора

Головні

- Запис інформації у пам'ять мікросхеми
- Зчитування інформації із пам'яті мікросхеми
- Перевірка правильності записаної інформації

Допоміжні

- Допомога у створенні програмного забезпечення (перевірка та модифікація змінних, забезпечення точок зупинки програми тощо)
- Створення одного або кількох каналів для обміну даними між мікросхемою та комп'ютером

Програмне забезпечення для мікроконтролерів зазвичай створюється в інтегрованому середовищі розробки (Integrated Development Environment, IDE). Типове IDE містить редактор коду, компілятор, та засоби оптимізації створеного коду. Дуже часто до складу такого середовища входять симулятори, що дозволяють протестувати роботу програмного забезпечення без завантаження його в мікроконтролер. Крім того, більшість IDE підтримують інструменти для завантаження програмного коду в мікроконтролер з використанням поширених програматорів.

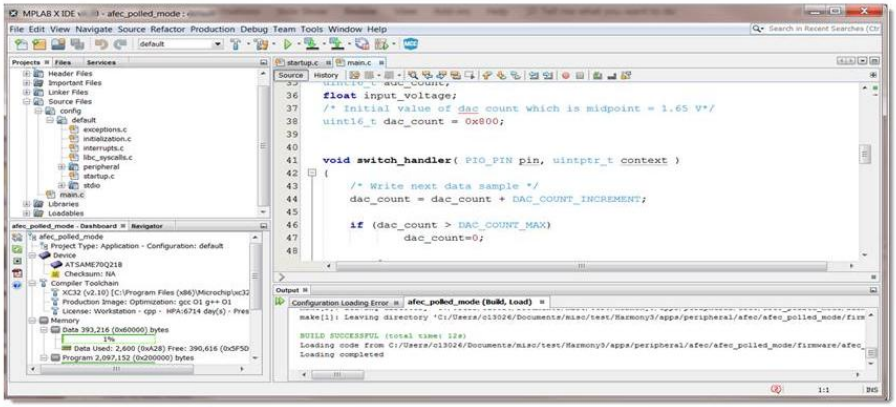
Інтегроване середовище розробки

Інтегроване середовище розробки (Integrated Development Environment, IDE) — комплексне програмне рішення для розробки програмного забезпечення

- Редактори коду
- Компілятори
- Засоби оптимізації
- Симулятори
- Підтримка програматорів
- Підтримка внутрішньосхемного налагодження
- Засоби керування проєктами
- Підтримка хмарних технологій

Сучасні IDE можуть інтегрувати хмарні сервіси, що дозволяє зберігати проєкти онлайн, виконувати компіляцію на сервері або використовувати віддалені симулятори. Класичний інтерфейс включає менеджер проєктів, список файлів, редактор коду та вікно для відображення результатів компіляції. Для фахівців з інформаційних технологій такий інтерфейс вже виглядає знайомо, адже він повторює логіку роботи з програмним забезпеченням для звичайних комп'ютерних застосунків.

Інтегроване середовище розробки



Таким чином, програмування мікроконтролерів сьогодні є актуальним напрямком у сфері інформаційних технологій, який відкриває широкі можливості для подальшого працевлаштування та забезпечує конкурентну заробітну плату, зокрема й в Україні, в якій використання мікроконтролерів, що є основною для дронів та інших безпілотних засобів, є одним із самих затребуваних сфер працевлаштування.

Сьогодні ви дізналися, що для початку роботи з мікроконтролерами потрібно мати три основні речі, які утворюють так званий «стартовий набір»: плату для розробки, програматор та інтегроване середовище для створення програмного забезпечення. Цей набір дозволяє не лише писати та компілювати програми, а й завантажувати їх у мікроконтролер та перевіряти працездатність у реальному часі. Використання стартового набору значно спрощує перші кроки в програмуванні та розробці електронних пристроїв, мінімізуючи ризик помилок через неправильне підключення компонентів або неповну схему.

Висновки

- Програмування мікроконтролерів є актуальним IT-напрямком з можливістю подальшого працевлаштування із високою заробітною платнею (у тому числі і в Україні)
- Для початку роботи з мікроконтролерами необхідно мати «стартовий набір»: плата для розробки, програматор, інтегроване середовище для розробки програмного забезпечення
- Виробники мікроконтролерів забезпечують комплексну підтримку, розробників, що використовують їх продукти

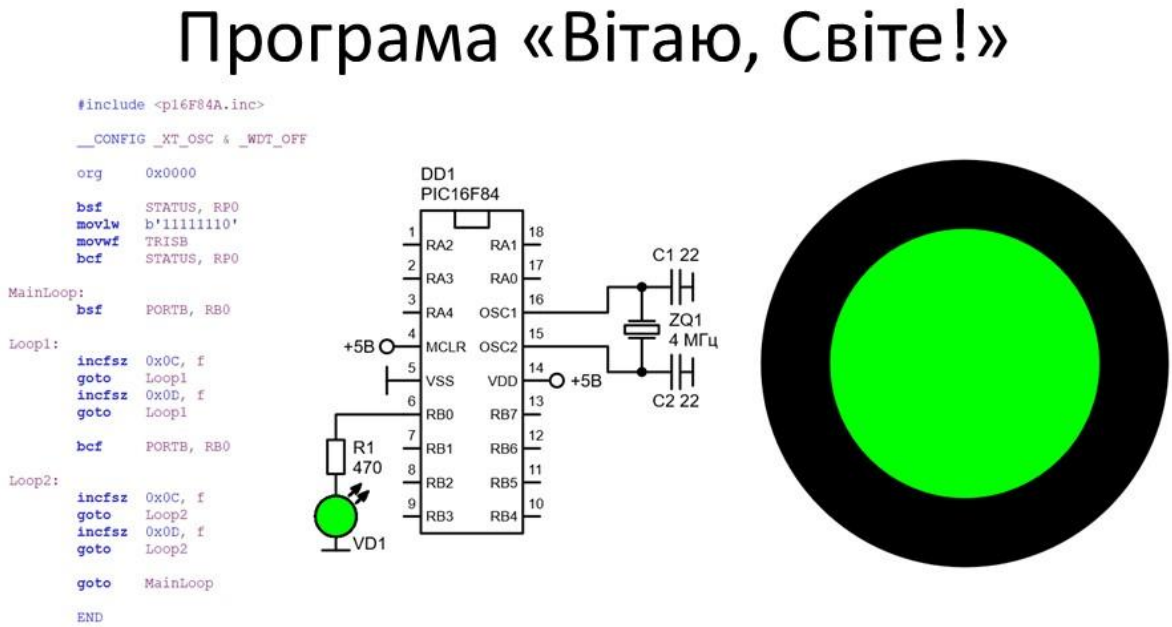
І не слід забувати про те, що виробники мікроконтролерів забезпечують комплексну підтримку розробників, що використовують їх продукцію, надаючи не лише самі мікросхеми, а й широкий спектр допоміжних засобів та програмних інструментів. До цього належать платформи для оцінки та розробки, стартові набори з усією необхідною периферією, інтегровані середовища розробки з компіляторами та симуляторами, приклади готових програм та опорні проекти з відкритими вихідними кодами. Таке комплексне забезпечення дозволяє швидко перевіряти працездатність компонентів, тестувати алгоритми, навчатися ефективному використанню мікроконтролерів і прискорює розробку нових пристроїв без потреби починати проєктування «з нуля».

Отже сьогодні ми з вами зробили тільки перший крок у світ мікроконтролерів. А далі ми будемо у цей світ потроху занурюватися. І це відбудеться зовсім скоро – вже на першому практичному занятті вам доведеться встановити середовище для розробки програмного забезпечення написати та перевірити першу програму, яка на жаргоні розробників електроніки називається «Вітаю, світе!».

ПРОГРАМА «ВІТАЮ, СВІТЕ!»

На минулому тижні ви познайомилися з мікроконтролерами, встановили інтегроване середовище для розробки програмного забезпечення MPLAB, написали першу програму для мікроконтролера PIC16F84, а також перевірили її працездатність у віртуальній лабораторії PIC_Lab. Якщо ви цього ще й досі не зробили, то потрібно це зробити якнайшвидше, бо інакше просто не будете розуміти речі, про які буде йти мова на наступних заняттях.

Ще раз нагадую, що мову асемблеру потрібно опановувати покроково, поступово підвищуючи рівень складності питань. На минулому тижні ми зробили підготовчі операції – встановили усі необхідні інструменти для роботи та перевірили їх працездатність. Настав час рухатись далі і зрозуміти, як працює програма, яку ми створили. Тому сьогодні ми докладно розберемо кожен рядок та кожну літеру у написаному вами вихідному коді, а вже на наступному практичному завданні ви напишете вашу першу самостійну програму, яка буде вирішувати ваше перше технічне завдання – автомат світлових ефектів.



Та перед тим як перейти до розбору безпосередньо програмного забезпечення спершу потрібно познайомитись з документом, яким ви будете активно користуватися впродовж усього курсу. Цей документ офіційно називається «Технічна документація на мікроконтролер PIC16F84», а на жаргоні розробників електроніки та програмістів мікроконтролерів цей документ називається скорочено «Datasheet». Технічна документація на будь-які електронні компоненти, зокрема і на мікроконтролери, розроблюється виробниками електронних компонентів, які найкраще за всіх знають як вони працюють. Тому коли у вас виникають будь-які питання, по роботі мікроконтролера, то перше джерело, де потрібно шукати відповіді – це технічна документація. Не ChatGPT, не сайти в Інтернеті, не книжки, не статті, не друзі чи одногрупники, а тільки технічна документація повинна бути вашим першим джерелом інформації при створенні програмного забезпечення.

Мікроконтролер PIC16F84 має нетиповий склад технічної документації – для цього мікроконтролера усі інформація, необхідна для роботи, зібрана в одному документі, який містить усього 88 сторінок. Для сучасних мікроконтролерів це дуже мало. Типова документація на сучасні мікроконтролери займає тисячі сторінок, в яких розробнику програмного забезпечення треба вміти швидко орієнтуватися.

Технічна документація (DataSheet)

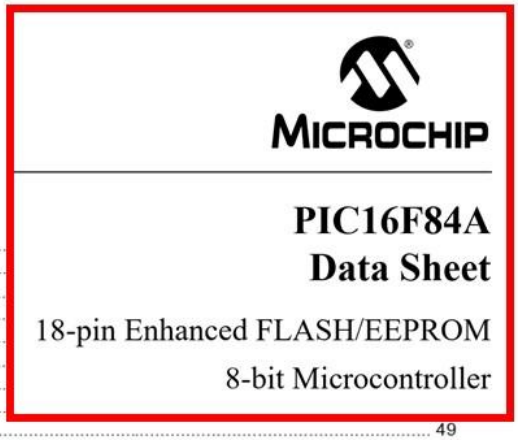


Table of Contents

1.0 Device Overview
2.0 Memory Organization
3.0 Data EEPROM Memory
4.0 I/O Ports
5.0 Timer0 Module
6.0 Special Features of the CPU
7.0 Instruction Set Summary
8.0 Development Support
9.0 Electrical Characteristics 49
10.0 DC/AC Characteristic Graphs 61
11.0 Packaging Information 71
Appendix A: Revision History 75
Appendix B: Conversion Considerations 76
Appendix C: Migration from Baseline to Mid-Range Devices 78
Index 79
On-Line Support 83
Reader Response 84
PIC16F84A Product Identification System 85

Самим великим за обсягом документом, який входить до переліку технічної документації, зазвичай, є опис лінійки або сімейства мікроконтролерів (Reference Manual). У цьому документі докладно розписані усі вузли та периферійні модулі мікроконтролерів, які входять до складу цієї групи. Для потужних розвинених сімейств мікроконтролерів документ, що описує усі їх можливості, може займати кілька тисяч сторінок. Проте, на відміну від літературних творів, які потрібно читати від початку до кінця, технічну документацію не потрібно читати повністю. Ви звертаєтесь до технічної документації, коли у вас є певне конкретне питання, наприклад, «В якому регістрі відбувається налаштування переривань?». У цьому випадку ви шукаєте в документації розділ «Переривання» (Interrupts), де і знаходите вичерпну відповідь на ваше питання. Якщо розробники вважають, що для кращого розуміння вам потрібно прочитати інші розділи, то там будуть відповідні посилання, з якими, на думку розробників мікроконтролера, вам також бажано, але необов'язково, ознайомитись.

Отже, Reference Manual містить інформацію про всі вузли, які можуть бути в групі однотипних мікроконтролерів. Але в конкретній моделі мікроконтролера можуть бути присутніми далеко не всі вузли, що є лінійці. Тому до переліку обов'язкової технічної документації входить ще один документ, який називається «Datasheet». У цьому документі наведено перелік вузлів та периферійних модулів, що входять до складу конкретної мікросхеми, в також у усі її унікальні особливості, наприклад, реальна кількість каналів введення-виведення, реальна кількість каналів АЦП та інша інформація.

Крім Reference Manual та Datasheet розробнику також бажано ознайомитись із документом, який містить відомі помилки у роботі конкретної мікросхеми, який називається «Errata». Тут не треба дивуватись – мікроконтролери також створюють люди, які також можуть щось не врахувати на етапі підготовки до серійного виробництва мікросхем. Якщо ці помилки наведені в Errata, то це означає, що вони стали відомими вже після того, як виготовлено мільйони мікросхем, які розійшлись по всьому світу. У цьому випадку виробник в Errata визначає, що та чи інша функція у цій мікросхемі працює не так, як описано в технічній документації, і наводить рекомендації по уникненню неправильної роботи програмного забезпечення.

Перелік технічної документації, що використовується при роботі з мікроконтролерами

RM0016
Reference manual
STM8S Series and STM8AF Series 8-bit microcontrollers

Introduction

This reference manual provides complete information for application developers on how to use STM8S Series and STM8AF Series microcontroller memory and peripherals.

The STM8AF Series of microcontrollers is designed for automotive applications, with different memory densities, packages and peripherals:

- The low-density STM8AF devices are the STM8AF623/26 with 8 Kbytes of Flash memory.
- The medium-density STM8AF devices are the STM8AF624x and STM8AF626/628 microcontrollers with 16 to 32 Kbytes of Flash memory.
- The high-density STM8AF devices are the STM8AF52xx and STM8AF626/8/9/Ax/Ex microcontrollers with 32 to 128 Kbytes of Flash memory.

The STM8S Series of microcontrollers is designed for general purpose applications, with different memory densities, packages and peripherals.

- The value-line low-density STM8S devices are the STM8S001xx/STM8S003xx microcontrollers with 8 Kbytes of Flash memory.
- The value-line medium-density STM8S devices are the STM8S005xx microcontrollers with 32 Kbytes of Flash memory.
- The value-line high-density STM8S devices are the STM8S007xx microcontrollers with 64 Kbytes of Flash memory.
- The access-line low-density STM8S devices are the STM8S103xx and STM8S903xx microcontrollers with 8 Kbytes of Flash memory.
- The access-line medium-density STM8S devices are the STM8S105xx microcontrollers with 16 to 32 Kbytes of Flash memory.
- The performance-line high-density STM8S devices are the STM8S207xx and STM8S208xx microcontrollers with 32 to 128 Kbytes of Flash memory.

Refer to the product datasheet for ordering information, pin description, mechanical and electrical device characteristics, and for the complete list of available peripherals.

Reference documents

- For information on programming, erasing and protection of the internal Flash memory please refer to the STM8S Series and STM8AF Series Flash programming manual (PM0051) and to the STM8 SWIM communication protocol and debug module user manual (UM0470).
- For information on the STM8 core, refer to STM8 CPU programming manual (PM0044).
- The bootloader user manual (UM0560) describes the usage of the integrated ROM bootloader.

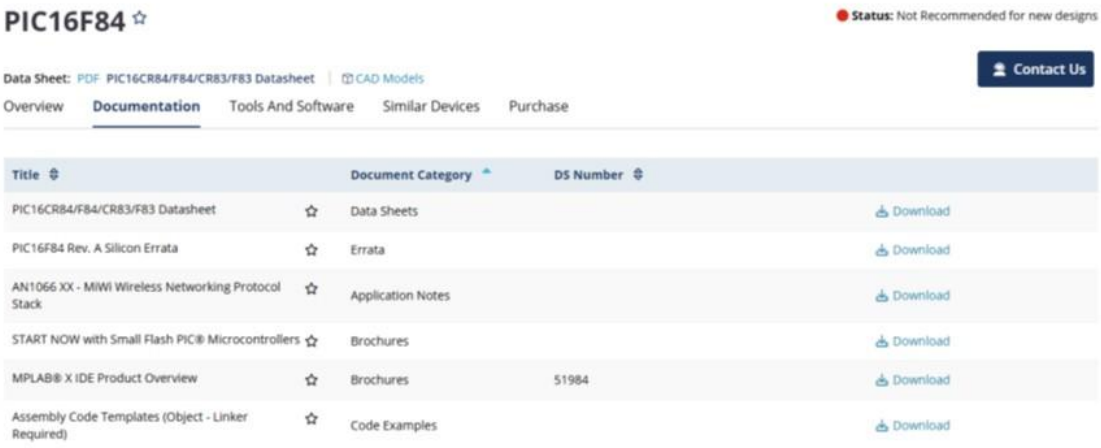
October 2017 DocID14587 Rev 14 1/487

- **Reference Manual** – технічна документація та лінійку (сімейство) мікроконтролерів (до 10 000 сторінок)
- **DataSheet** – технічна документація на конкретну мікросхему (до 300 сторінок)
- **Errata** – документ, що містить відомі помилки в роботі мікроконтролера
- **Інші документи**

Окрім цих трьох основних документів: Reference Manual, Datasheet та Errata, можуть знадобиться і інші документи, наприклад рекомендації по використанню того чи іншого периферійного модуля. Але кожен розробник, що працює з мікроконтролерами, повинен завжди мати під рукою три основних документа, щоб не гаяти час на пошук відповідей на питання, які постійно виникають впродовж розробки програмного забезпечення.

Технічну документацію на мікроконтролер потрібно брати з сайту виробника. При цьому слід слідкувати за версіями файлів і використовувати в роботі лише найновіші актуальні версії. Технічна документація на мікроконтролери загального призначення зазвичай знаходиться у вільному доступі і доступна розробникам на безоплатній основі.

Технічна документація на сайті виробника


PIC16F84

Status: Not Recommended for new designs

Contact Us

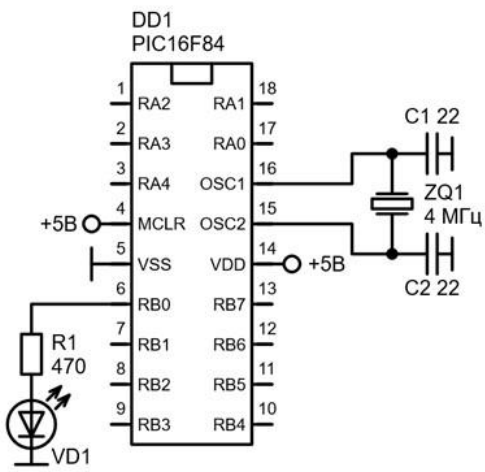
Data Sheet: PDF PIC16CR84/F84/CR83/F83 Datasheet | CAD Models

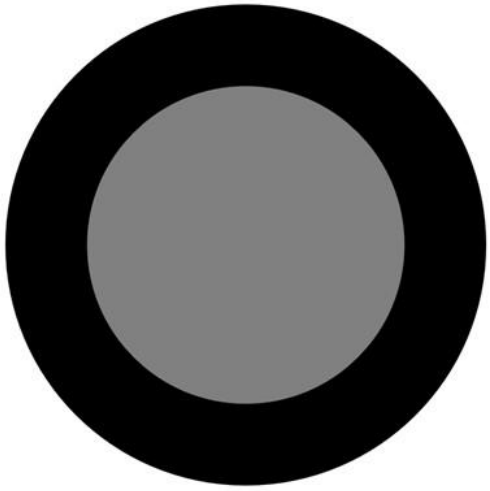
Overview | Documentation | Tools And Software | Similar Devices | Purchase

Title	Document Category	DS Number	
PIC16CR84/F84/CR83/F83 Datasheet	Data Sheets		Download
PIC16F84 Rev. A Silicon Errata	Errata		Download
AN1066 XX - MiWi Wireless Networking Protocol Stack	Application Notes		Download
START NOW with Small Flash PIC® Microcontrollers	Brochures		Download
MPLAB® X IDE Product Overview	Brochures	51984	Download
Assembly Code Templates (Object - Linker Required)	Code Examples		Download

Тепер, коли ви знаєте де брати технічну документацію на мікроконтролер, давайте розберемося з електричною схемою. Відкриваємо наш лабораторний макет і намагаємось зрозуміти, що ми бачимо, і як усе це працює. Звертаю увагу, що багато питань з функціонування електричної частини схем на основі мікроконтролерів ми з вами вивчали в курсі «Електротехніка та електроніка», тому, можливо, вам доведеться знову відкрити цей курс та згадати деякі питання.

Електрична схема





Головним елементом схеми є мікроконтролер, що має унікальний номер DD1. Ви пам’ятаєте, що на електричних схемах кожен електронний компонент для однозначної ідентифікації повинен мати унікальний номер, який складається з букв, що визначають тип електронного компонента (цифрові мікросхеми, зокрема мікроконтролери, позначаються літерами «DD») та цифр.

На нашій схемі умовне зображення мікроконтролера є прямокутником, який повторює зовнішній вигляд найбільш популярного корпусу мікросхеми – DIP18, що має 18 контактів, розташованих зі стандартним кроком 2,54 мм по обидва боки прямокутного пластикового корпусу. Біля прямокутника розташовується унікальний ідентифікатор мікроконтролера (DD1) та модель (PIC16F84 або PIC16F84A).

На одній із коротких сторін корпусу розташований ключ, що має форму точки або виїмки. Цей ключ позначає місце, де розташований перший вивід мікросхеми (це правило стосується усіх мікросхем – не тільки мікросхем мікроконтролерів). Нумерація виводів мікросхем йде проти годинникової стрілки, якщо дивитись на мікросхему зверху. На електричних схемах розташування виводів майже завжди не відповідає їх фізичному розташуванню на корпусі, тому поруч із виводом завжди позначають його номер, який вказують ззовні прямокутника безпосередньо біля виводу.

Умовне позначення мікроконтролера

Всередині умовного позначення мікроконтролера для полегшення сприйняття схеми вказують функцію відповідного виводу. В мікроконтролерах до кожного з виводів може бути

підключено декілька периферійних вузлів, тому більшість з виводів є багатофункціональними. Наприклад, відповідно до технічної документації, в мікроконтролерах PIC16F84 до третього виводу мікросхеми підключений четвертий канал порту введення-виведення A (RA4) та тактовий вхід таймеру TMR0 (T0CKI). На електричних схемах через брак місця зазвичай вказують лише одну найбільш поширену функцію. Але для програмістів та електронників навіть ця стисла ця інформація є вкрай важливою, оскільки вона зв'язує апаратну та програмну частини проєкту та дозволяє швидко зорієнтуватися у схемі.

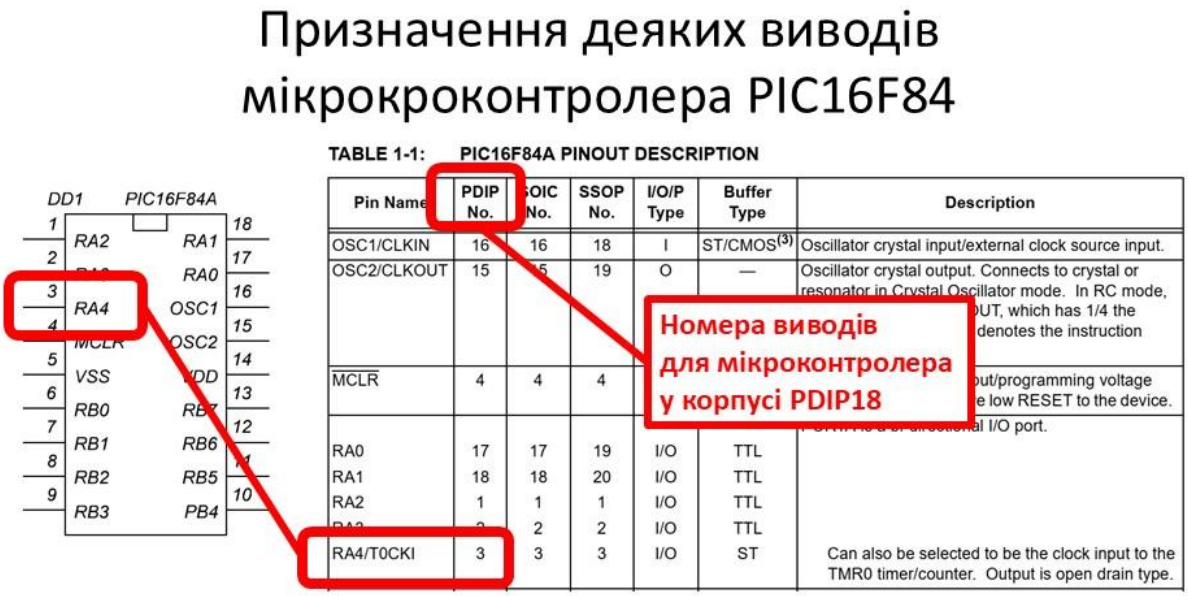


TABLE 1-1: PIC16F84A PINOUT DESCRIPTION

Pin Name	PDIP No.	SOIC No.	SSOP No.	I/O/P Type	Buffer Type	Description
OSC1/CLKIN	16	16	18	I	ST/CMOS(3)	Oscillator crystal input/external clock source input.
OSC2/CLKOUT	15	15	19	O	—	Oscillator crystal output. Connects to crystal or resonator in Crystal Oscillator mode. In RC mode, OUT, which has 1/4 the denotes the instruction
MCLR	4	4	4			ut/programming voltage e low RESET to the device.
						al I/O port.
RA0	17	17	19	I/O	TTL	
RA1	18	18	20	I/O	TTL	
RA2	1	1	1	I/O	TTL	
RA3	2	2	2	I/O	TTL	
RA4/T0CKI	3	3	3	I/O	ST	Can also be selected to be the clock input to the TMR0 timer/counter. Output is open drain type.

Усі виводи мікроконтролера умовно поділяються на дві категорії: загального та спеціального призначення. Виводи загального призначення не мають якогось заздалегідь відомого функціонального призначення і визначаються застосунком, в якому цей мікроконтролер буде використовуватися. Виводи спеціального призначення мають певне функціональне призначення, яке зазвичай є принципово необхідним для працездатності мікроконтролера, тому вони майже завжди підключаються до певних ділянок або вузлів електричної схеми.

В мікроконтролерах PIC16F84 до виводів спеціального призначення відносяться:

– вивід VSS, призначений для підключення до спільного проводу схеми, інколи його позначають як GND – скорочення від англійського слова «Ground» (земля);

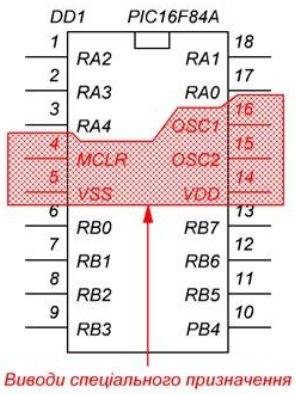
– вивід VDD, призначений для подачі живлення на мікроконтролер, який інколи позначають як VCC;

– вивід MCLR (Master Clear), призначений для апаратного перезавантаження мікроконтролера, який також часто позначають як RST, або NRST, що є скороченням від англійського слова «Reset» (перезавантаження);

– виводи OSC1, OSC2, зв'язані з вбудованим тактовим генератором – абревіатура виводів є скороченням від англійського слова «Oscillator» (генератор).

Усі інші виводи (RA0 – RA4, RB0 – RB7) є виводами загального призначення, функція яких визначається конкретним застосунком.

Призначення виводів мікроконтролера



Виводи спеціального призначення:

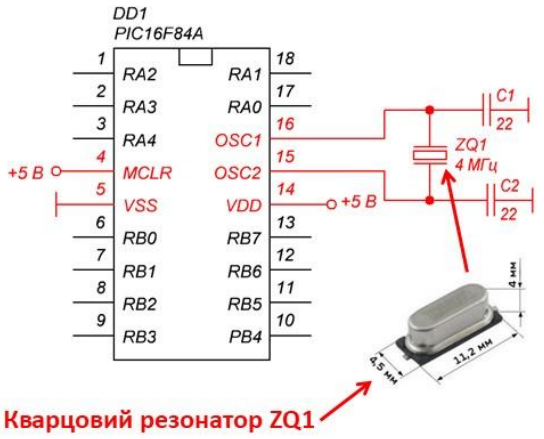
- **VSS** – підключення до спільного проводу схеми
- **VDD** – підключення до джерела живлення (+2…5,5 В)
- **MCLR** (Master Clear) – вивід апаратного перезавантаження мікроконтролера (активний рівень – рівень логічного нуля)
- **OSC1, OSC2** – виводи, зв'язані з тактовим генератором

Виводи загального призначення:

- **RA0 – RA4** – виводи, зв'язані з портом введення-виведення PORTA (5 каналів)
- **RB0 – RB7** – виводи, зв'язані з портом введення-виведення PORTB (8 каналів)

І тепер стає зрозумілим підключення певних ділянок нашої електричної схеми: вивід VSS, призначений для підключення до спільного проводу, підключений до спільного проводу, вивід VDD, призначений для подачі живлення, підключений до джерела живлення з напругою +5 В, вивід MCLR, призначений для апаратного перезавантаження мікроконтролера у нашій схемі не використовується, тому він підключений до джерела живлення, що відповідає подачі на цей вивід сигналу з рівнем логічної одиниці (+5 В), а до виводів OSC1 та OSC2, відповідно до технічної документації, підключений зовнішній кварцовий резонатор ZQ1 з резонансною частотою 4 МГц та блокувальні конденсатори C1 та C1, що мають ємність 22 пФ.

Підключення виводів спеціального призначення



- **VSS** – до спільного проводу
- **VDD** – до джерела живлення (+5 В)
- **MCLR** – не використовується (подано сигнал, з рівнем логічної одиниці +5 В)
- **OSC1, OSC2** – підключені до кварцового резонатора ZQ1 з резонансною частотою 4 МГц, відповідно до технічної документації

Така схема включення мікроконтролера PIC16F84 є типовою і повністю відповідає технічній документації. Ця частина схеми буде майже однаковою у всіх проєктах. Єдине що може відрізнятися – так це частота та тип резонатора, яка буде визначати з якою швидкістю наш мікроконтролер буде працювати. Цей параметр ми можемо обирати самостійно, відповідно до нашого технічного завдання. Частіше за все мікроконтролери PIC16F84 працюють на частотах 20 МГц, 4 МГц або 32768 Гц, проте частота може бути і іншою. Ємність блокувальних конденсаторів C1 та C2 залежить від частоти резонатора і обирається відповідно до технічної документації на мікросхему.

Рекомендації по вибору блокувальних конденсаторів C1 та C2

FIGURE 6-1: CRYSTAL/CERAMIC RESONATOR OPERATION (HS, XT OR LP OSC CONFIGURATION)

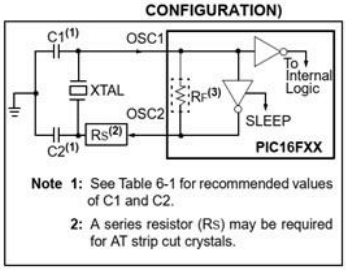


TABLE 6-1: CAPACITOR SELECTION FOR CERAMIC RESONATORS

Ranges Tested:			
Mode	Freq	OSC1/C1	OSC2/C2
XT	455 kHz 2.0 MHz 4.0 MHz	47 - 100 pF 15 - 33 pF 15 - 33 pF	47 - 100 pF 15 - 33 pF 15 - 33 pF
HS	8.0 MHz 10.0 MHz	15 - 33 pF 15 - 33 pF	15 - 33 pF 15 - 33 pF

Note: Recommended values of C1 and C2 are identical to the ranges tested in this table. Higher capacitance increases the stability of the oscillator, but also increases the start-up time. These values are for design guidance only. Since each resonator has its own characteristics, the user should consult the resonator manufacturer for the appropriate values of external components.

TABLE 6-2: CAPACITOR SELECTION FOR CRYSTAL OSCILLATOR

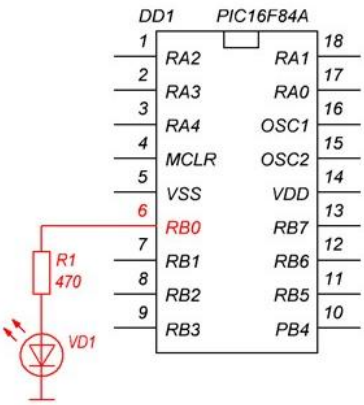
Mode	Freq	OSC1/C1	OSC2/C2
LP	32 kHz 200 kHz	68 - 100 pF 15 - 33 pF	68 - 100 pF 15 - 33 pF
XT	100 kHz 2 MHz 4 MHz	100 - 150 pF 15 - 33 pF 15 - 33 pF	100 - 150 pF 15 - 33 pF 15 - 33 pF
HS	4 MHz 20 MHz	15 - 33 pF 15 - 33 pF	15 - 33 pF 15 - 33 pF

Note: Higher capacitance increases the stability of the oscillator, but also increases the start-up time. These values are for design guidance only. Rs may be required in HS mode, as well as XT mode, to avoid over-driving crystals with low drive level specification. Since each crystal has its own characteristics, the user should consult the crystal manufacturer for appropriate values of external components.
For V_{DD} > 4.5V, C1 = C2 ≈ 30 pF is recommended.

Тепер перейдемо до унікальної частини електричної схеми, яка є лише у цьому проєкті. А вона складається лише з двох електронних компонентів: індикаторного світлодіода VD1, який випромінює світло при протіканні електричного струму, та струмообмежувального резистора R1 з опором 470 Ом, який обмежує його струм.

Тепер настав час розібратися з роботою. А для цього нам потрібно спершу познайомитися з найпоширенішими периферійними модулями, які є в кожному мікроконтролері. Ці периферійні модулі називаються «порти введення-виведення».

Унікальна частина проєкту



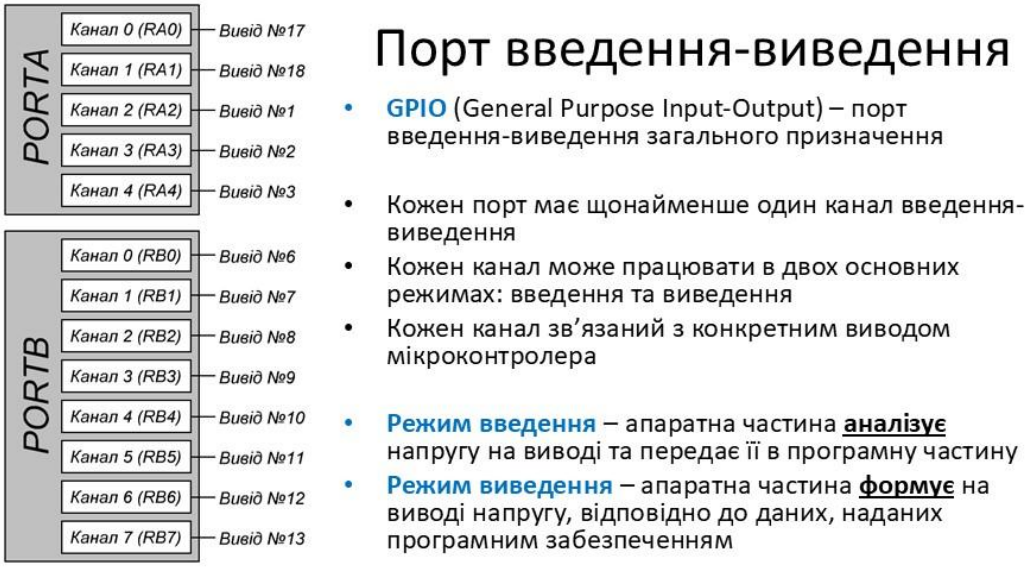
- **VD1** – індикаторний світлодіод
- **R1** – струмообмежувальний резистор з опором 470 Ом

Індикаторні світлодіоди

Порти введення-виведення загального призначення (General Purpose Input-Output, GPIO) є найпоширенішими периферійними модулями сучасних мікроконтролерів. Мікроконтролер PIC16F84 має два порти введення-виведення: A та B. Кожен порт повинен мати щонайменше один канал введення-виведення, який електрично зв'язаний з конкретним виводом мікроконтролера. У мікроконтролері PIC16F84 порт A має 5 каналів, а порт B – 8 каналів. Канали порту можуть бути однакові, але зазвичай усі канали різні, тому треба уважно вивчати технічну документацію конкретної моделі мікроконтролера, щоб чітко собі уявляти особливості та обмеження кожного з каналів.

Кожен канал порті введення-виведення може працювати в двох основних режимах: введення та виведення. У режимі введення апаратна частина каналу порту аналізує напругу

на виводі та передає її в програмну частину. В режимі виведення усе відбувається навпаки – апаратна частина формує напругу на виводі мікроконтролера, відповідно до інформації, встановленої програмним забезпеченням.



За одну лекцію я просто фізично не зможу пояснити усі особливості роботи портів введенні-виведення в різних режимах, тому давайте зараз просто проаналізуємо нашу схему і зупинимося на поясненні лише тих питань, які нам потрібно з'ясувати тут і зараз. А аналіз електричної схеми нашого першого проєкту дає нам наступні результати.

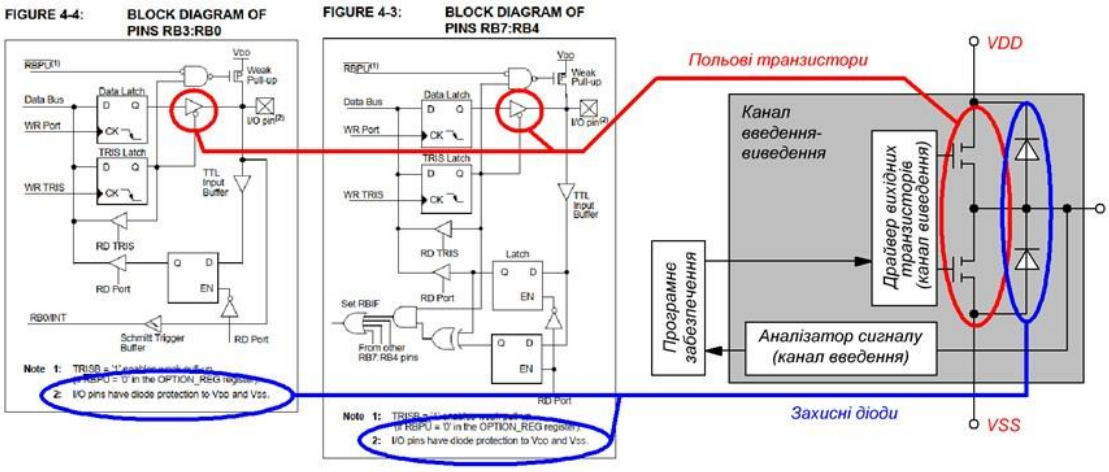
1. Індикаторний світлодіод VD1 підключений до каналу 0 порту B (RB0), тому надалі нас буде цікавити лише цей канал. Усі інші канали портів введення-виведення у цьому проєкті не використовуються.

2. Нашим світлодіодом потрібно керувати. Для цього необхідно формувати на виводі RB0 якісь напруги. Це значить, що канал RB0 повинен працювати в режимі виведення. Це головне, що нам зараз потрібно знати. Тепер давайте розбиратися, як нам це зробити.

Проаналізуємо спрощену електричну схема каналів порту B, яка наведена в технічній документації на мікроконтролер. Кожен канал можна умовно поділити на дві частини: частину, що відповідає за аналіз напруги на виводі мікроконтролера, тобто за роботу в режимі введення, і частину, що відповідає за формування напруги на виводі мікроконтролера, тобто за роботу в режимі виведення.

Нас цікавить робота в режимі виведення. В цьому режимі головними елементами є два потужні польові транзистори, які можуть з'єднати вивід мікроконтролера (вивід RB0) або з шиною живлення (вивід VDD), або зі спільним проводом (вивід VSS). Транзисторами керує спеціальна апаратна частина яка називається «драйвер». Драйвер формує напруги керування які можуть перевести кожен транзистор або в стан з високим, або в стан з малим внутрішнім опором.

Порт введення-виведення

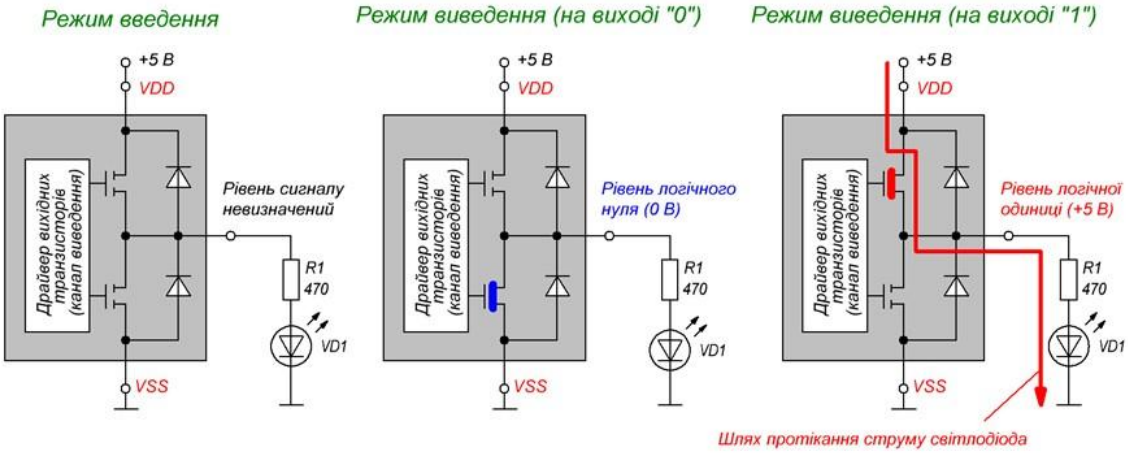


Таким чином, наша схема може знаходитися у трьох можливих станах. Коли канал RB0 знаходиться у стані введення, обидва вихідні транзистори каналу знаходяться у стані з високим внутрішнім опором. У цьому стані напруга на виводі мікроконтролера не визначається програмним забезпеченням – вона цілком і повністю визначається зовнішньою електричною схемою. У цьому стані коло виводу RB0 розімкнене, тому світлодіод VD1 світити не буде.

Налаштуємо канал RB0 для роботи у режим виведення і переведемо його в стан формування на виводі сигналу з рівнем логічного нуля. Це призведе до того, що нижній (за схемою) транзистор перейде у стан з малим, а верхній – у стан з великим внутрішнім опором. За такої комутації нижній транзистор підключить вивід мікроконтролера до спільного проводу (з'єднає між собою виводи RB0 та VSS). У цьому випадку утвориться замкнене електричне коло, але в ньому не буде джерел електричної енергії. Тому у цьому стані струм через світлодіод VD1 протікати не буде, і світла випромінювати він не буде.

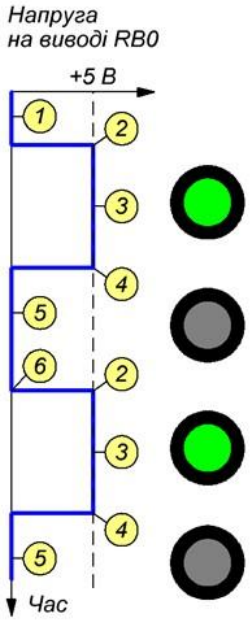
Налаштуємо канал RB0 для роботи у режим виведення і переведемо його в стан формування на виводі сигналу з рівнем логічної одиниці. Це призведе до того, що нижній (за схемою) транзистор перейде у стан з великим, а верхній – у стан з малим внутрішнім опором. За такої комутації верхній транзистор підключить вивід мікроконтролера до шини живлення (з'єднає між собою виводи RB0 та VDD). У цьому випадку утвориться замкнене електричне коло, частиною якого буде джерело електричної живлення мікроконтролера. Це призведе до того, що через світлодіод VD1 почне протікати струм, що буде обмежуватись резистором R1, і він почне випромінювати світло.

Принцип керування світлодіодом



Отже тепер ми можемо чітко сформулювати, алгоритм програмного забезпечення, яке на потрібно розробити. Спочатку нам потрібно перевести канал RB0 в режим виведення. Потім засвітити наш світлодіод VD1, сформувавши на виводі RB0 сигнал з рівнем логічної одиниці (+5 В). Після цього потрібно зачекати 0,2…0,5 с, а потім загасити світлодіод VD1, сформувавши на виводі RB0 сигнал з рівнем логічного нуля (0 В). Після цього потрібно знову зачекати 0,2…0,5 с, а потім перейти до пункту 2. Бачите, що наш алгоритм не має кінця. Світлодіод VD1 тепер буде блимати допоки мікроконтролер підключений до джерела живлення.

Тепер, нарешті, ми закінчили з розглядом апаратної частини і можемо переходити до самої цікавої частини нашого курсу – розробки програмного забезпечення.



Алгоритм програмного забезпечення

1. Перевести канал RB0 в режим виведення
2. Сформувати на виводі RB0 сигнал з рівнем логічної одиниці (+5 В)
3. Зачекати 0,2...0,5 с
4. Сформувати на виводі RB0 сигнал з рівнем логічного нуля (0 В)
5. Зачекати 0,2...0,5 с
6. Перейти до пункту 2

А як програмістам керувати апаратною частиною мікроконтролера? Ви вже маєте певний досвід програмування на мовах високого рівня і добре знаєте що таке змінні. Хто забув, або ще й досі не знає, то нагадую, що змінна – це одна або кілька комірок оперативної пам’яті, в яких можна зберігати інформацію. Зазвичай в комп’ютерах оперативна пам’ять

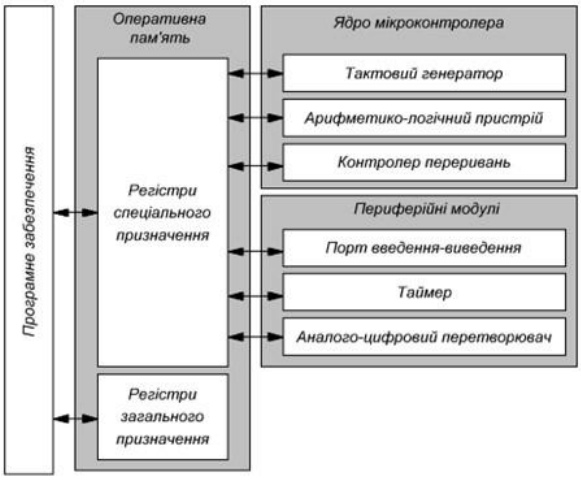
використовується для зберігання довільної інформації, але в мікроконтролерах вбудована оперативна пам'ять використовується дещо інакше.

В мікроконтролерах комірки оперативної пам'яті поділяються на дві частини: комірки спеціального призначення, та комірки загального призначення. Комірки загального призначення використовуються як традиційна оперативна пам'ять – для зберігання довільної інформації, тому вони змінюються виключно програмним шляхом.

Комірки спеціального призначення, з точки зору програміста, нічим не відрізняються від комірок загального призначення. Програміст може працювати з ними так само, як і з комірками загального призначення. Однак на відміну від комірок загального призначення, комірки спеціального призначення мають зв'язок з апаратними вузлами мікроконтролера: ядром та периферійними модулями. Тому інформація в них може змінюватися як програмним, так і апаратним шляхом. Більше того, програмна зміна інформації в комірках спеціального призначення призводить до зміни роботи апаратної частини мікроконтролера.

Таким чином, з точки зору програміста, керування вузлами мікроконтролера зводиться до маніпуляції з інформацією, розташованою в комірках спеціального призначення. Тобто програміст працює з апаратною частиною як зі звичайними змінними, які можна читати та записувати. Єдина відмінність полягає у тому, що ці змінні повинні мати фіксовані адреси в оперативній пам'яті, але ця особливість жодним чином не впливає на швидкість написання програмного коду.

Принцип керування апаратною частиною мікроконтролера



Регістри (комірки) оперативної пам'яті спеціального призначення мікроконтролера PIC16F84

- **TMR0** – регістр таймеру
- **OPTION_REG** – різні налаштування системи
- **PCL** – програмний лічильник (молодший байт)
- **PCLATH** – програмний лічильник (старший байт)
- **STATUS** – прапори арифметико-логічних операцій
- **FSR** – регістр непрямої адресації
- **PORTA** – стан порту A
- **TRISA** – режим роботи порту A
- **PORTB** – стан порту B
- **TRISB** – режим роботи порту B
- **EEDATA** – дані пам'яті EEPROM
- **EEADR** – адреса активної комірки пам'яті EEPROM
- **EECON1** – налаштування пам'яті EEPROM
- **EECON2** – налаштування пам'яті EEPROM
- **INTCON** – налаштування переривань

З усього переліку регістрів спеціального призначення мікроконтролера нас зараз цікавлять лише ті, які пов'язані із портом B. В мікроконтролері PIC16F84 основна інформація, пов'язана з цим портом, зберігається в двох регістрах: TRISB та PORTB. Кожен з цих регістрів є 8-розрядним, тобто кожен регістр має 8 біт. Кожен біт регістру зв'язаний з певним каналом порту: нульовий біт пов'язаний з нульовим каналом, перший біт – з першим каналом, другий біт – з другим каналом і, врешті-решт, сьомий біт зв'язаний сьомим каналом.

Регістр TRISB відповідає за режими роботи каналів. Якщо біт встановлений в одиницю, то цей канал порту буде працювати в режимі введення, а якщо в нуль – то цей канал буде працювати в режимі виведення.

Регістр PORTB відповідає за стан каналу. Якщо канал працює в режимі виведення, то програмне встановлення цього біту в значення «0» призведене до формування на відповідному виводі мікроконтролера напруги з рівнем, що відповідає рівню логічного нуля, а якщо в цей біт записати «1», то на виході буде сформований сигнал з рівнем, що відповідає рівню логічної одиниці.

В режимі введення, коли відповідний біт у регістрі TRISB встановлений в одиницю, стан біту в регістрі PORTB визначається напругою на виводі мікроконтролера. Якщо рівень напруги інтерпретується апаратною частиною як рівень логічного нуля, то цей біт буде апаратно очищений, тобто встановлений в значення «0». А якщо напруга на виводі буде мати значення достатнє для інтерпретації сигналу, як сигналу з рівнем логічної одиниці, то цей біт буде апаратно встановлений в значення «1».

Регістри, що керують портом PORTB

TRISB

Біт 7	Біт 6	Біт 5	Біт 4	Біт 3	Біт 2	Біт 1	Біт 0
Режим роботи каналу RB7	Режим роботи каналу RB6	Режим роботи каналу RB5	Режим роботи каналу RB4	Режим роботи каналу RB3	Режим роботи каналу RB2	Режим роботи каналу RB1	Режим роботи каналу RB0

PORTB

Біт 7	Біт 6	Біт 5	Біт 4	Біт 3	Біт 2	Біт 1	Біт 0
Стан каналу RB7	Стан каналу RB6	Стан каналу RB5	Стан каналу RB4	Стан каналу RB3	Стан каналу RB2	Стан каналу RB1	Стан каналу RB0

Світлодіод VD1 підключений до каналу RB0, тому нас цікавлять лише наймолодші нульові біти в регістрах TRISB та PORTB. Особливістю регістру TRISB, як і усіх інших регістрів, що відповідають за режими роботи портів введення-виведення, є те, що після перезавантаження мікроконтролера УСІ канали УСІХ портів введення-виведення примусово налаштовуються в режим введення. Таким чином, після перезавантаження мікроконтролера біт 0 у регістрі TRISB буде встановлений в значення «1». У цьому режимі наш світлодіод ніколи не буде світити, тому нам потрібно перевести цей канал порту в режим виведення. Це – перший крок нашого алгоритму.

Після цього, ми засвічуємо світлодіод VD1, встановивши біт 0 у регістрі PORTB рівним «1». При цьому вивід мікроконтролера, зв’язаного з цим каналом, буде підключено до шини живлення, на ньому з’явиться напруга +5 В, під дією якої через світлодіод VD1 стане протікати струм і він почне випромінювати світло.

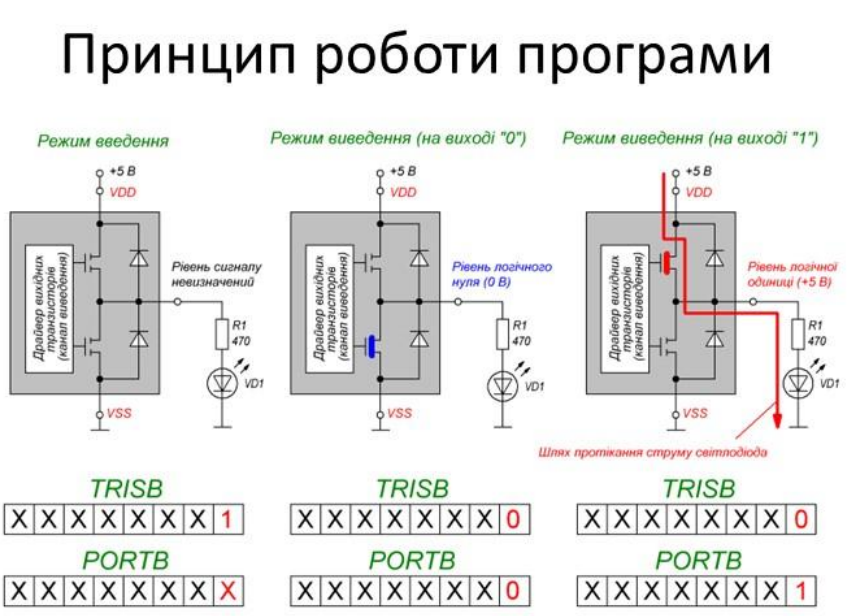
Після цього треба деякий час (0,2…0,5 секунд) зачекати – щоб наше око встигло побачити світло.

Потім ми записуємо у біт 0 регістру PORTB значення «0», що призводить до відключення виводу мікроконтролера від шини живлення і підключення його до спільного проводу. За такої комутації напруга на виводі мікроконтролера стане рівною 0 В, струм через світлодіод VD1 перестане протікати, і він згасне.

Після цього ми знову чекаємо деякий час (0,2…0,5 секунд) – щоб наше око встигло адаптуватися, а потім зациклюємо наш алгоритм шляхом переходу до пункту 2.

Отже тепер ми розуміємо загальне призначення усі логічних блоків нашого вихідного коду. Що вони роблять і навіщо вони потрібні. Настав час розібратися якими засобами вони були реалізовані.

```
        org     0x0000

;       1. Записати 0 у біт 0
;       регістру TRISB
        bsf     STATUS, RP0
        movlw   b'11111110'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
;       2. Записати 1 у біт 0
;       регістру PORTB
        bsf     PORTB, RB0

;       3. Зачекати 0,2…0,5 с
Loop1:
        incfsz  0x0C, f
        goto    Loop1
        incfsz  0x0D, f
        goto    Loop1

;       4. Записати 0 у біт 0
;       регістру PORTB
        bcf     PORTB, RB0

;       5. Зачекати 0,2…0,5 с
Loop2:
        incfsz  0x0C, f
        goto    Loop2
        incfsz  0x0D, f
        goto    Loop2

;       6. Перейти до пункту 2
        goto    MainLoop
```



Для початку зауважимо, що якщо ви забудете як пишеться, або як працює та чи інша асемблерна команда, то нічого страшного не станеться – у вас під рукою завжди є технічна документація на мікроконтролер, де докладно описані усі 35 асемблерних команд, які реалізовані в мікроконтролері PIC16F84. Ці команди ми будемо вивчати поступово, тому не треба сидіти і примусово їх вчити. З часом ви їх усі запам’ятаєте і будете використовувати навіть не усвідомлюючи, що ви їх знаєте.

TABLE 7-2: PIC16CXXX INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	00	0001	1fff	ffff	Z	2
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1 (2)	00	1011	dfff	ffff		1,2,3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1 (2)	00	1111	dfff	ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	f	Move W to f	1	00	0000	1fff	ffff		
NOP	-	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2

А першою асемблерною командою, з якою ви познайомитесь, буде команда goto. Команда goto є командою безумовного переходу до команди, що має в пам’яті програм адресу, вказану в цій команді.

Давайте подивимось, як компілятор розмістив вашу програму в пам’яті програм. Першу асемблерну команду, яку ви написали, компілятор розмістив у пам’яті програм за адресою 0x0000. Як він це зробив? А ви самі вказали компілятору початкову адресу директивою «org». Ця директива прямо вказує компілятору, в якій частині пам’яті програм потрібно розміщувати наступні команди.

Комірка пам’яті програм, що має адресу 0x0000 є особливою коміркою. Вона називається «Вектор перезавантаження» (Reset Vector). Коли мікроконтролер виходить із стану перезавантаження, то першою командою, яку він виконає, буде команда, розташована в комірці пам’яті програм з адресою 0x0000.

У нормальному режимі роботи ядро мікроконтролера виконує команди по черзі. Тобто спочатку виконається команда за адресою 0x0000, потім 0x0001, потім 0x0002 і так допоки на закінчиться пам’ять програм. Мікроконтролер PIC16F84 має 1024 комірки пам’яті програм, таким чином остання комірка буде мати адресу 0x03FF. Після цієї комірки ядро знову почне виконувати команди спочатку, тобто з адреси 0x0000. Але так буде лише в нормальному режимі.

Серед переліку команд є команди, які порушують цей поряд. Однією з них і є команда goto. Якщо у комірці пам’яті буде команда goto, то наступною командою, яку виконає мікроконтролер буде не команда, що розташована в наступній комірці, а команда, розташована в комірці, адреса якої вказана в команді goto.

Зверніть увагу, що команда goto працює з фізичними адресами пам’яті програм, тобто з числами. А нам визначити ці адреси? Суто теоретично, адресу комірки пам’яті, куди треба перейти, можна розрахувати, і в команді goto вказати конкретне число. Але так не роблять навіть найзухваліші фанати програмування на мовах асемблеру. Річ у тому, що будь-яка модифікація програмного забезпечення, наприклад, додавання додаткової команди на початку роботи програми, призведе до необхідності переобчислення усіх адрес, що є в програмі.

Для вирішення цієї проблеми в мовах програмування використовують мітки. Мітка – це унікальний набір символів, який розташовується на самому початку рядку. Хорошим зразком програмування є закінчення мітки двокрапкою, що показує, що це саме мітка, а не випадковий набір символів. Через необхідність використання міток, оператори, тобто команди, у коді пишуть не з початку рядка, а на деякій відстані – зазвичай 8 символів.

У нашій програмі є три команди, що мають мітки. Саме на ці команди і посилаються команди goto. При створенні програми компілятор самостійно визначає за якими адресами будуть розташовані мічені команди, а потім, також самостійно, замінює мітки конкретними адресами.

Усі наші безумовні переходи показані на рисунку стрілками. Ми поки не будемо розбирати як працюють переходи на команди з мітками Loop1 та Loop2. Зараз дивимося на глобальний перехід, який виконує остання команда goto, що посилається на командою з міткою MainLoop. Це глобальний нескінченний цикл нашої програми.

Тепер ми бачимо, що у нашій програмі задіяні лише 12 комірок пам’яті програм, і що завдяки наявності останнього безумовного переходу команди, що розташовані в комірках з адресами більшими за 0x000C, ніколи виконуватися на будуть.

```
        org     0x0000

;       1. Записати 0 у біт 0
;       регістру TRISB
        bsf     STATUS, RP0
        movlw   b'11111110'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
;       2. Записати 1 у біт 0
;       регістру PORTB
        bsf     PORTB, RB0

;       3. Зачекати 0,2…0,5 с
Loop1:
        incfsz  0x0C, f
        goto    Loop1
        incfsz  0x0D, f
        goto    Loop1

;       4. Записати 0 у біт 0
;       регістру PORTB
        bcf     PORTB, RB0

;       5. Зачекати 0,2…0,5 с
Loop2:
        incfsz  0x0C, f
        goto    Loop2
        incfsz  0x0D, f
        goto    Loop2

;       6. Перейти до пункту 2
        goto    MainLoop
```

Алгоритм програмного забезпечення

1. Перевести канал RB0 в режим виведення
2. Сформувати на виводі RB0 сигнал з рівнем логічної одиниці (+5 В)
3. Зачекати 0,2…0,5 с
4. Сформувати на виводі RB0 сигнал з рівнем логічного нуля (0 В)
5. Зачекати 0,2…0,5 с
6. **Перейти до пункту 2**

Отже ми розібралися як працює Пункт 6 нашого алгоритму. Тепер давайте подивимось як реалізовані пункти 2 та 4. Як ви вже напевно здогадалися, вони реалізовуються майже ідентично.

Команди для роботи з бітами в регістрах оперативної пам’яті

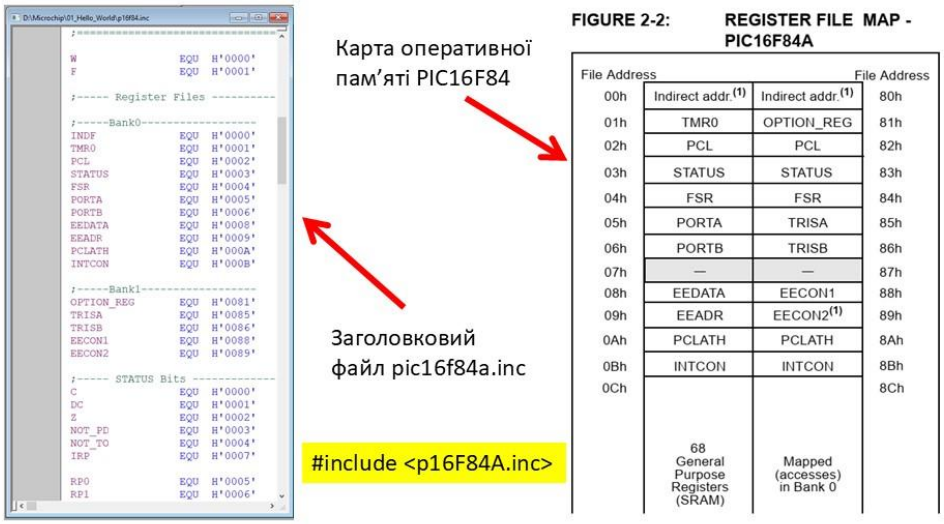
BCF	**Bit Clear f**
Syntax:	[*label*] BCF f,b
Operands:	$0 \le f \le 127$ $0 \le b \le 7$
Operation:	$0 \rightarrow$ (f<b>)
Status Affected:	None
Description:	Bit 'b' in register 'f' is cleared.

BSF	**Bit Set f**
Syntax:	[*label*] BSF f,b
Operands:	$0 \le f \le 127$ $0 \le b \le 7$
Operation:	$1 \rightarrow$ (f<b>)
Status Affected:	None
Description:	Bit 'b' in register 'f' is set.

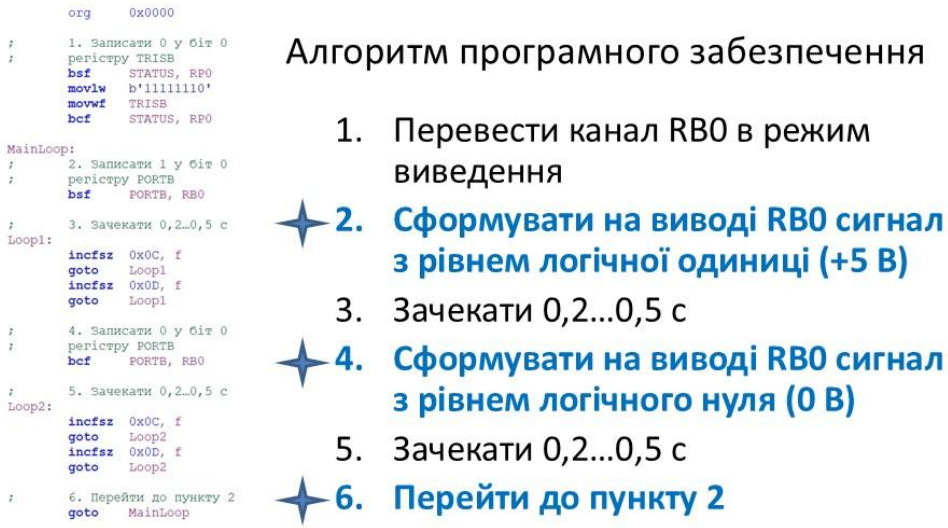
Адреса	*Мітка*	*Команда*	*Команда*
0x0000		*bsf STATUS, RP0*	*bsf 0x03, 5*
0x0001		*movlw b'11111110'*	*movlw 0xFE*
0x0002		*movwf TRISB*	*movwf 0x86*
0x0003		*bcf STATUS, RP0*	*bcf 0x03, 5*
0x0004	*MainLoop:*	*bsf PORTB, RB0*	*bsf 0x06, 0*
0x0005	*Loop1:*	*incfsz 0x0C, f*	*incfsz 0x0C, 1*
0x0006		*goto Loop1*	*goto 0x005*
0x0005		*incfsz 0x0D, f*	*incfsz 0x0D, 1*
0x0006		*goto Loop1*	*goto 0x005*
0x0007		*bcf PORTB, RB0*	*bcf 0x06, 0*
0x0008	*Loop2:*	*incfsz 0x0C, f*	*incfsz 0x0C, 1*
0x0009		*goto Loop2*	*goto 0x0008*
0x000A		*incfsz 0x0D, f*	*incfsz 0x0D, f*
0x000B		*goto Loop2*	*goto 0x0008*
0x000C		*goto MainLoop*	*goto 0x0004*

Змінити певний біт в певному регістрі оперативної пам’яті можна за допомогою команд bcf та bsf. Команда bcf встановлює біт у значення «0», команда bsf – у значення «1». При використанні цих команд потрібно вказати адресу комірки оперативної пам’яті, яка буде модифікована, та номер біта – від 0 до 7.

Але зверніть увагу, що в нашому коді ми не вказуємо реальну адресу, і не вказуємо реальний номер біта, хоча так також можна робити. Замість цього ми вказуємо їх назви: назву регістра спеціального призначення (PORTB) та назву біту його нульового каналу (RB0). Але як компілятор розуміє, що таке PORTB та RB0?



А цю інформацію компілятор отримує з заголовкового файлу «pic16f84a.inc», який ми підключили у першому рядку нашої програми за допомогою директиви «#include». У цьому файлі знаходяться найбільш поширені константи, зокрема і назви регістрів спеціального призначення з вказанням їх фізичних адрес. Ці назви узгоджені з технічною документацією, що дозволяє нам краще розуміти, що ми робимо. Коли компілятор зустрічає замість числа якусь назву, то він переглядає файл pic16f84a.inc, і підставляє замість неї число, що їй відповідає. Наявність подібних файлів не тільки прискорює написання коду, а ще й спрощує міграцію коду між різними моделями мікроконтролерів. Наприклад, в одному мікроконтролері регістр PORTB має адресу 0x06, а в іншому – 0x07. У цьому випадку достатньо замінити заголовковий файл «xxx.inc» та перекомпілювати проєкт, і тепер ваша програма буде працювати на іншій апаратній основі.



Отже ми розібралися з половиною пунктів нашого алгоритму. Йдемо далі.

Давайте тепер подивимось як ми налаштовуємо канал RB0 для роботи в режимі виведення. Для цього нам потрібно встановити біт 0 (він називається TRISB0) в регістрі TRISB у значення «0». Для цього можна скористатися вже відомою нам командою bcf. Проте краще це зробити іншим способом.

Річ у тому, що на етапі налаштування периферійних модулів мікроконтролера, краще налаштовувати весь периферійний модуль, ніж його частину. Тому ми у цій частині коду налаштовуємо одразу усі канали порту TRISB. Нульовий канал ми налаштовуємо для роботи в режимі виведення, а усі інші – в режим введення.

Це робиться за допомогою двох асемблерних команд: movlw та movwf. Перша команда (movlw) завантажує числову константу у спеціальний регістр ядра, який називається

акумулятором (аналог надшвидкої кеш-пам’яті в сучасних процесорах). А друга копіює інформацію з акумулятора в комірку оперативної пам’яті, адреса якої вказана в команді.

У нашому проєкті ми записуємо в регістр TRISB значення b'11111110' – це число записане у двійковому форматі, про що свідчить префікс «b» на початку запису (скорочення від «binary»). Двійковий формат дозволяє нам наочно побачити стан кожного біту, що дуже зручно при роботі з регістрами, що є бітовими мапами (бітова мапа – число, де кожен розряд має унікальне призначення). Тепер ми добре бачимо, що нульовий канал порту B налаштовується в режим виведення, а усі інші – в режим введення.

Налаштування регістру TRISB

MOVLW	**Move Literal to W**
Syntax:	[*label*] MOVLW k
Operands:	$0 \le k \le 255$
Operation:	$k \rightarrow (W)$
Status Affected:	None
Description:	The eight-bit literal 'k' is loaded into W register. The don't cares will assemble as 0's.

MOVWF	**Move W to f**
Syntax:	[*label*] MOVWF f
Operands:	$0 \le f \le 127$
Operation:	$(W) \rightarrow (f)$
Status Affected:	None
Description:	Move data from W register to register 'f'.

Адреса	*Мітка*	*Команда*	*Команда*
0x0000		*bsf STATUS, RP0*	*bsf 0x03, 5*
0x0001		*movlw b'11111110'*	*movlw 0xFE*
0x0002		*movwf TRISB*	*movwf 0x86*
0x0003		*bcf STATUS, RP0*	*bcf 0x03, 5*
0x0004	*MainLoop:*	*bsf PORTB, RB0*	*bsf 0x06, 0*
0x0005	*Loop1:*	*incfsz 0x0C, f*	*incfsz 0x0C, 1*
0x0006		*goto Loop1*	*goto 0x005*
0x0005		*incfsz 0x0D, f*	*incfsz 0x0D, 1*
0x0006		*goto Loop1*	*goto 0x005*
0x0007		*bcf PORTB, RB0*	*bcf 0x06, 0*
0x0008	*Loop2:*	*incfsz 0x0C, f*	*incfsz 0x0C, 1*
0x0009		*goto Loop2*	*goto 0x0008*
0x000A		*incfsz 0x0D, f*	*incfsz 0x0D, f*
0x000B		*goto Loop2*	*goto 0x0008*
0x000C		*goto MainLoop*	*goto 0x0004*

Але перед цими командами є одна незрозуміла команда «bsf STATUS, RP0». Ми вже знаємо, що команда bsf призначена для встановлення біту у комірці оперативної пам’яті у значення «1». Тобто перед тим як записати значення у регістр TRISB нам чомусь закортіло встановити біт RP0 у регістрі STATUS у значення «1». Навіщо?

Тут є одна особливість мікроконтролерів PIC середнього (і не тільки) сімейства, до якого відноситься і мікроконтролер PIC16F84. Формат зберігання команд у пам’яті програм цих мікроконтролерів дозволяє записати лише 7 бітів адреси оперативної пам’яті. Для того щоб не обмежувати максимально можливий обсяг оперативної пам’яті 128 байтами, розробники пішли наступним шляхом. При зверненні до оперативної пам’яті фізична адреса комірки утворюється наступним шляхом: молодші 7 бітів беруться з коду команди, а старші – з регістру STATUS.

В мікроконтролері PIC16F84 старший восьмий біт адреси оперативної пам’яті – це біт RP0 у регістрі STATUS. Таким чином, щоб прочитати чи записати щось у комірку оперативної пам’яті, що має адресу 0x00 – 0x7F, потрібно спочатку встановити біт RP0 рівним 0, а при звернені до регістрів з адресами 0x80 – 0xFF – рівним 1. Після перезавантаження мікроконтролера біт RP0 у регістрі STATUS апаратно встановлюється у значення «0», а комірки спеціального призначення в оперативній пам’яті мікроконтролера розташовані таким чином, що комірки, до яких частіше за все звертається програмне забезпечення, мають адреси 0x00 – 0x7F. Усе це зроблено для того, що зменшити кількість перемикань біту RP0. Крім того, самі важливі регістри спеціального призначення (STATUS, PCL, FSR тощо) мають дві адреси, що відрізняються старшим восьмим бітом, наприклад, регістр STATUS має адреси: 0x03 (b’00000011’) та 0x83 (b’10000011’). Тобто самі виробники добре розуміють, що така система формування адреси є недосконалою, однак напевно, коли розроблювався мікроконтролер іншого виходу в них не було.

REGISTER 2-1: STATUS REGISTER (ADDRESS 03h, 83h)

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

bit 7-6 **Unimplemented:** Maintain as '0'

bit 5 **RP0**: Register Bank Select bits (used for direct addressing)
01 = Bank 1 (80h - FFh)
00 = Bank 0 (00h - 7Fh)

bit 4 $\overline{TO}$: Time-out bit
1 = After power-up, CLRWDT instruction, or s
0 = A WDT time-out occurred

FIGURE 2-2: REGISTER FILE MAP - PIC16F84A

File Address			File Address
00h	Indirect addr.(1)	Indirect addr.(1)	80h
01h	TMR0	OPTION_REG	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h

Біт RP0 у регістрі STATUS встановлює активну сторінку оперативної пам'яті. Щоб звертатися до регістрів, що мають адресу 0x00 – 0x7F, потрібно, щоб цей біт був встановлений в 0, а при звернені до регістрів з адресами 0x80 – 0xFF – рівним 1

Отже перед тим як записати значення у регістр TRISB, що має адресу 0x86, ми спочатку встановлюємо біт RP0 у регістрі STATUS рівним «1», потім записуємо потрібне значення у регістр TRISB, а потім одразу повертаємо біт RP0 у регістрі STATUS у початкове значення «0», щоб уникнути подальшої плутанини. Якщо цього не зробити, то замість регістру TRISB, що має адресу 0x86 (b'10000110') інформація запишеться у регістр PORTB, що має адресу 0x06 (b'00000110').

Налаштування регістру TRISB

- Встановити біт RP0 у регістрі STATUS у значення 1
- Записати в акумулятор константу b'11111110'
- Скопіювати зміст акумулятора в регістр TRISB
- Повернути біт RP0 у регістрі STATUS у початкове значення

Адреса	*Мітка*	*Команда*	*Команда*
0x0000		*bsf STATUS, RP0*	*bsf 0x03, 5*
0x0001		*movlw b'11111110'*	*movlw 0xFE*
0x0002		*movwf TRISB*	*movwf 0x86*
0x0003		*bcf STATUS, RP0*	*bcf 0x03, 5*
0x0004	*MainLoop:*	*bsf PORTB, RB0*	*bsf 0x06, 0*
0x0005	*Loop1:*	*incfsz 0x0C, f*	*incfsz 0x0C, 1*
0x0006		*goto Loop1*	*goto 0x005*
0x0005		*incfsz 0x0D, f*	*incfsz 0x0D, 1*
0x0006		*goto Loop1*	*goto 0x005*
0x0007		*bcf PORTB, RB0*	*bcf 0x06, 0*
0x0008	*Loop2:*	*incfsz 0x0C, f*	*incfsz 0x0C, 1*
0x0009		*goto Loop2*	*goto 0x0008*
0x000A		*incfsz 0x0D, f*	*incfsz 0x0D, f*
0x000B		*goto Loop2*	*goto 0x0008*
0x000C		*goto MainLoop*	*goto 0x0004*

Це є особливістю мікроконтролерів PIC. Її потрібно просто знати і враховувати при написанні програмного коду, бо інакше наша програма працювати просто не буде. Тепер ми розібралися ще з одним пунктом нашого алгоритму і залишилося тільки зрозуміти як нам робити затримку. Найпростішим способом формування затримки у часі є завантаження мікроконтролера порожньою роботою. У даному випадку ми можемо використовувати одну з двох цікавих команд: incfsz або decfsz. У нашому проєкті була використана команда incfsz. Давайте подивимось, як це працює.

```
        org     0x0000

;       1. Записати 0 у біт 0
;       регістру TRISB
        bsf     STATUS, RP0
        movlw   b'11111110'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
;       2. Записати 1 у біт 0
;       регістру PORTB
        bsf     PORTB, RB0

;       3. Зачекати 0,2...0,5 с
Loop1:
        incfsz  0x0C, f
        goto    Loop1
        incfsz  0x0D, f
        goto    Loop1

;       4. Записати 0 у біт 0
;       регістру PORTB
        bcf     PORTB, RB0

;       5. Зачекати 0,2...0,5 с
Loop2:
        incfsz  0x0C, f
        goto    Loop2
        incfsz  0x0D, f
        goto    Loop2

;       6. Перейти до пункту 2
        goto    MainLoop
```

Алгоритм програмного забезпечення

1. **Перевести канал RB0 в режим виведення**
2. **Сформувати на виводі RB0 сигнал з рівнем логічної одиниці (+5 В)**
3. Зачекати 0,2...0,5 с
4. **Сформувати на виводі RB0 сигнал з рівнем логічного нуля (0 В)**
5. Зачекати 0,2...0,5 с
6. **Перейти до пункту 2**

Спочатку ми збільшуємо на одиницю комірку пам’яті з адресою 0x0C. Це цією адресою знаходиться комірка загального призначення і в ній ми можемо зберігати будь-яку довільну інформацію. Далі ядро мікроконтролера перевіряє результат цього збільшення. Якщо у результаті збільшення число в комірці відрізняється від 0, то виконується наступна команда. Якщо ні, замість наступної команди виконується команда nop (немає операції), тобто наступна команда фактично пропускається. Наступною командою після incfsz є команда безумовного переходу goto, яка посилається на ту ж саму команду incfsz. Тобто ми, фактично зациклюємося у цій частині коду допоки в результаті збільшення на одиницю число в регістрі 0x0C не стане рівним нулю.

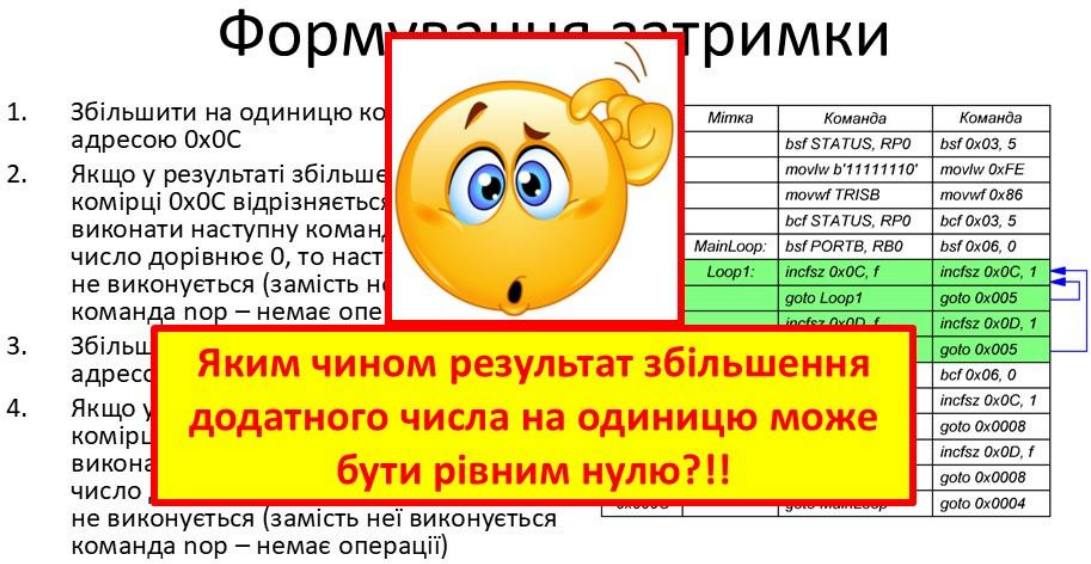
Після цього ми робимо аналогічні маніпуляції але вже з іншою коміркою, що має адресу 0x0D. Її ми так само будемо збільшувати її на одиницю допоки в результаті збільшення вона не стане рівною нулю. Тільки після цього ми перейдемо до наступних команд нашого алгоритму. Усі ці збільшення та переходи потребують певного часу, що і формує нам необхідну затримку.

Формування затримки

1. Збільшити на одиницю комірку пам’яті з адресою 0x0C
2. Якщо у результаті збільшення число в комірці 0x0C відрізняється від 0, то виконати наступну команду (goto), якщо число дорівнює 0, то наступна команда не виконується (замість неї виконується команда nop – немає операції)
3. Збільшити на одиницю комірку пам’яті з адресою 0x0D
4. Якщо у результаті збільшення число в комірці 0x0D відрізняється від 0, то виконати наступну команду (goto), якщо число дорівнює 0, то наступна команда не виконується (замість неї виконується команда nop – немає операції)

Адреса	*Мітка*	*Команда*	*Команда*
0x0000		*bsf STATUS, RP0*	*bsf 0x03, 5*
0x0001		*movlw b'11111110'*	*movlw 0xFE*
0x0002		*movwf TRISB*	*movwf 0x86*
0x0003		*bcf STATUS, RP0*	*bcf 0x03, 5*
0x0004	*MainLoop:*	*bsf PORTB, RB0*	*bsf 0x06, 0*
0x0005	*Loop1:*	*incfsz 0x0C, f*	*incfsz 0x0C, 1*
0x0006		*goto Loop1*	*goto 0x005*
0x0005		*incfsz 0x0D, f*	*incfsz 0x0D, 1*
0x0006		*goto Loop1*	*goto 0x005*
0x0007		*bcf PORTB, RB0*	*bcf 0x06, 0*
0x0008	*Loop2:*	*incfsz 0x0C, f*	*incfsz 0x0C, 1*
0x0009		*goto Loop2*	*goto 0x0008*
0x000A		*incfsz 0x0D, f*	*incfsz 0x0D, f*
0x000B		*goto Loop2*	*goto 0x0008*
0x000C		*goto MainLoop*	*goto 0x0004*

Цей алгоритм є цілком працездатним, проте у мене, особисто, виникає одне питання: яким чином результат збільшення на одиницю додатного числа може стати рівним нулю?



Ласкаво просимо у світ програмування, де не завжди працюють закони математики. Комірка оперативної пам'яті є 8-розрядною. Кожен розряд може приймати лише 2 значення: 0 або 1. Це означає, що комірка у цілому може мати лише $2^8 = 256$ комбінацій значень, які у найпростішому випадку інтерпретуються як додатні числа від 0 до 255. Таким чином, максимальне додатне число, яке може зберігатися у цій комірці пам'яті дорівнює 255. І якщо за допомогою команди incfsz збільшити на одиницю число 255 то відбудеться переповнення комірки, і число в ній стане рівним 0. Так само, якщо спробувати зменшити на одиницю комірку, що містить число 0, то замість від'ємного числа –1 у цій комірці буде збережено додатне число 255.

У цій частині лекції я завжди прошу студентів з теплотою загадати усіх ваших вчителів математики. Але у світі програмування арифметичні операції інколи виконуються по іншим правилам.

І на цій оптимістичній ноті ми і завершуємо нашу лекцію. Ми майже повністю розібрали увесь код нашої першої програми. А питання, які залишилися поза межами сьогоднішньої лекції, ми обов'язково розглянемо найближчим часом, коли будемо вирішувати наступні практичні задачі. А на сьогодні все. Дякую за увагу!

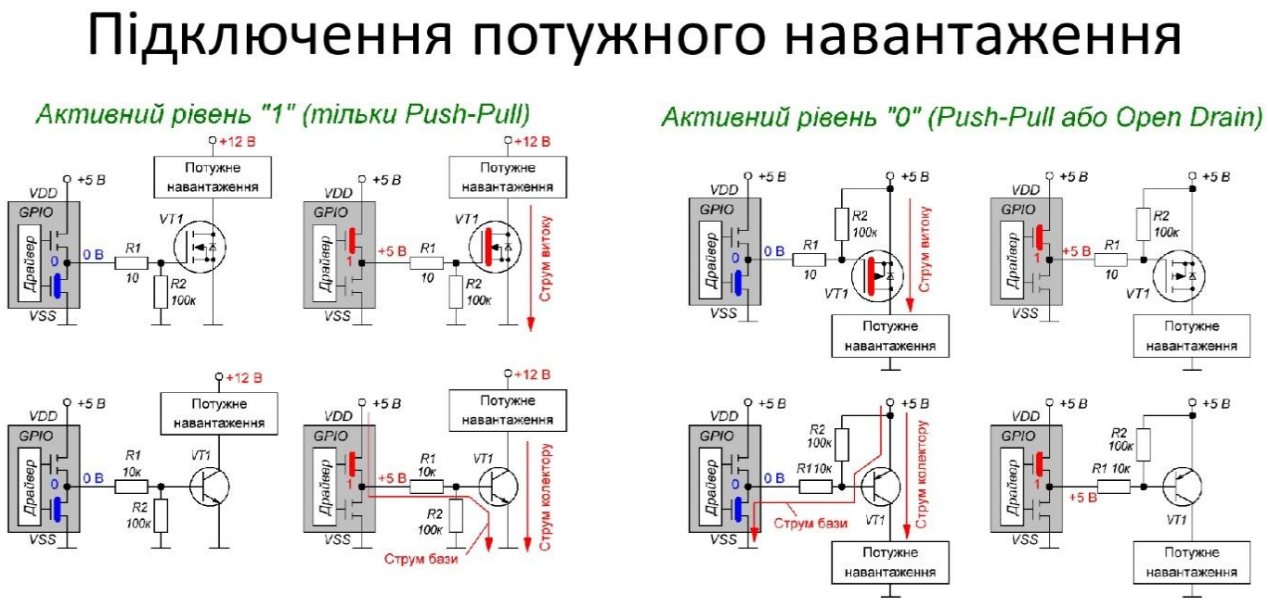
ПІДПРОГРАМИ ТА ЗАТРИМКИ

Сьогодні ми продовжуємо вивчати дисципліну «Програмування мікроконтролерів та пристроїв інтернет речей». На цій лекції ми дізнаємося, що таке підпрограми і затримки та як їх використовувати.

На минулій лекції ми дізналися, як написати свою першу програму «Вітаю, світе!», на принципі якої, можна програмувати велосипедні ліхтарики, світлофори, що блимають однією лампочкою, тощо. Також виконали роботу «Автомат світлових ефектів» – це ялинкові гірлянди, світло-музикальні установки, тощо.

Сьогодні ми дізнаємося, як виконати практичну роботу «Світлофор» – програму для керування дорожнім рухом. Завдання: запрограмувати світлофор так, щоб він відтворював реальні світлові сигнали. Для розуміння роботи рекомендується поспостерігати за реальним світлофором або прочитати правила дорожнього руху.

Для початку роботи, подивимось на принципову схему, на якій зображено шість каналів підсилювачів: дві червоні лампи, дві жовті, дві зелені. Лампочки працюють від 230 В, тому підключати їх напряму до мікроконтролера не можна. Для керування використовуються підсилювачі на польових транзисторах, які дозволяють керувати потужним навантаженням малопотужними сигналами.

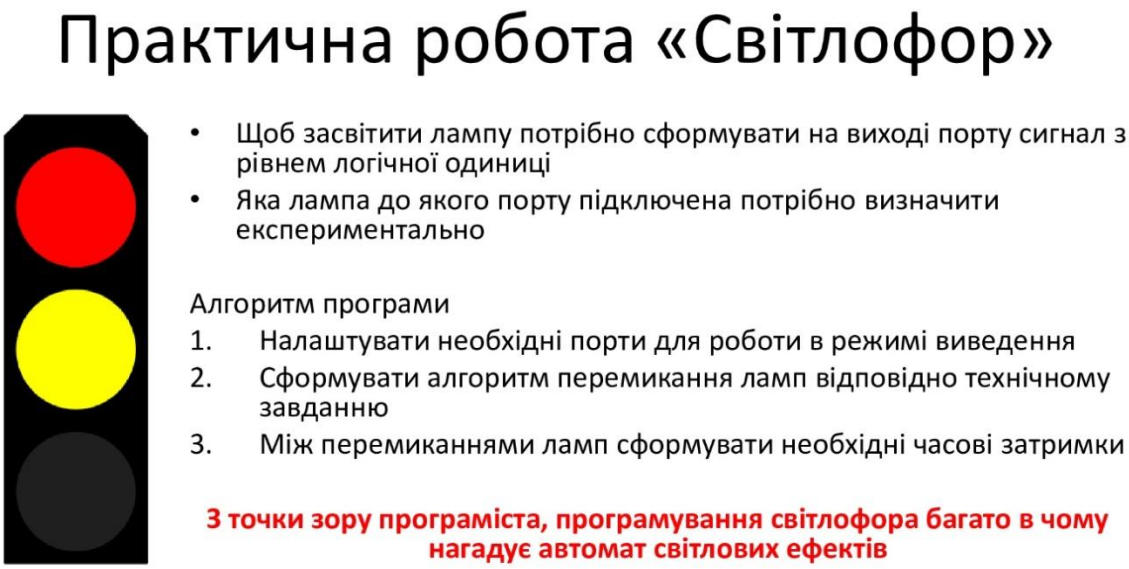


Для початку роботи давайте увімкнемо хоча б одну лампочку. Щоб це зробити необхідно подати на відповідний канал введення-виведення сигнал логічної одиниці. Якщо на каналі встановлений нуль, лампочка не світиться. Який саме канал керує конкретною лампочкою, необхідно визначити експериментально. У кожного варіанта лабораторної роботи своя схема, тому один і той самий канал у різних варіантах може керувати різними лампами або взагалі не бути підключеним.

Схеми підключення не надаються, тому визначення каналів виконується емпіричним методом. Потрібно послідовно подавати сигнал логічної одиниці на різні канали та спостерігати, яка лампочка засвітиться.

Отже, алгоритм роботи буде такий:

1. Визначити, які порти та канали підключені до кожної лампочки;
2. Налаштувати знайдені порти в режим виведення;
3. Формувати сигнали нуля та одиниці згідно з алгоритмом, який описаний у методичних матеріалах;
4. Додати часові затримки між перемиканнями, щоб світлові сигнали відповідали реальній роботі світлофора.



У попередній практичній роботі в методичних рекомендаціях був наведений приклад коду, який потрібно було реалізувати. У всіх варіантах код майже однаковий, відрізняється лише командами «movlw», де замість світлового ефекту потрібно було вставити власний варіант. Необхідно було реалізувати один світловий ефект, що складається з восьми фаз, тобто восьми комбінацій сигналів, які виводяться на порт B.

Після кожної зміни сигналів на порту B виконується затримка. При тактовій частоті 4 МГц її тривалість становить приблизно від 0,15 до 0,3 секунди. Затримка формується шляхом виконання великої кількості простих операцій: збільшення вмісту комірки пам’яті, повторення цього циклу 256 разів, а потім повторення всього блоку ще 256 разів. Загалом це близько 64 000 повторів плюс час, витрачений на переходи між командами.

Студенти, які писали програму на 90 балів, де таких блоків значно більше, відзначали складність у їх копіюванні та коригуванні. При великій кількості ефектів, це стає дуже незручним. Виділені ділянки коду є майже однаковими, за винятком того, що вони посилаються на різні мітки, які потрібно змінювати вручну.

Світловий ефект «Вогник, що біжить»

```
        org     0x000

        ; налаштування порту B
        bsf     STATUS, RP0
        movlw   b'00000000'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
        ; фаза 1
        movlw   b'00000001'
        movwf   PORTB

Loop1:  ; затримка (пауза)
        incfsz  0x0C
        goto    Loop1
        incfsz  0x0D
        goto    Loop1

        ; фаза 2
        movlw   b'00000010'
        movwf   PORTB

Loop2:  ; затримка (пауза)
        incfsz  0x0C
        goto    Loop2
        incfsz  0x0D
        goto    Loop2

        ; фаза 3
        movlw   b'00000100'
        movwf   PORTB

Loop3:  ; затримка (пауза)
        incfsz  0x0C
        goto    Loop3
        incfsz  0x0D
        goto    Loop3

        ; фаза 4
        movlw   b'00001000'
        movwf   PORTB

Loop4:  ; затримка (пауза)
        incfsz  0x0C
        goto    Loop4
        incfsz  0x0D
        goto    Loop4

        ; фаза 5
        movlw   b'00010000'
        movwf   PORTB

Loop5:  ; затримка (пауза)
        incfsz  0x0C
        goto    Loop5
        incfsz  0x0D
        goto    Loop5

        ; фаза 6
        movlw   b'00100000'
        movwf   PORTB

Loop6:  ; затримка (пауза)
        incfsz  0x0C
        goto    Loop6
        incfsz  0x0D
        goto    Loop6

        ; фаза 7
        movlw   b'01000000'
        movwf   PORTB

Loop7:  ; затримка (пауза)
        incfsz  0x0C
        goto    Loop7
        incfsz  0x0D
        goto    Loop7

        ; фаза 8
        movlw   b'10000000'
        movwf   PORTB

Loop8:  ; затримка (пауза)
        incfsz  0x0C
        goto    Loop8
        incfsz  0x0D
        goto    Loop8

        goto    MainLoop

        END
```

Для спрощення використання таких затримок, можна використовувати «лайфхак»: у невеликих фрагментах коду (до 10 команд), можна використовувати символ «$», замість міток. Долар позначає адресу поточної команди, і компілятор під час компіляції автоматично підставляє цю адресу. Наприклад, go to $-1 переходить на попередню команду, go to $-3 — на команду, що знаходиться на три позиції вище.

Код із мітками та код з використанням $ працюють однаково. Різниця полягає у відносній адресації, і зсуви доводиться рахувати вручну, тому цей лайфхак застосовують лише в невеликих ділянках коду для зручності та уникнення помилок.

Використання символу «$»

```
Loop1:  ; затримка (пауза)
        incfsz  0x0C
        goto    Loop1
        incfsz  0x0D
        goto    Loop1
```

→

```
        ; затримка (пауза)
        incfsz  0x0C
        goto    $ - 1
        incfsz  0x0D
        goto    $ - 3
```

$ – адреса комірки пам’яті програм, в якій розміщена поточна команда

Під час компіляції компілятор самостійно визначає адреси і підставляє в прошивку необхідні числа

$ – 1 – від поточної адреси віднімається 1 (на етапі компіляції)
$ – 3 – від поточної адреси віднімається 3 (на етапі компіляції)

goto $ – зациклення (зависання) програми

Таким чином ми отримуємо 8 однакових блоків коду, що значно спрощує написання коду та його коригування. Але чи є це гарним рішенням для програми така кількість однакових елементів?

Світловий ефект «Вогник, що біжить»

```
        org     0x000

        ; налаштування порту B
        bsf     STATUS, RP0
        movlw   b'00000000'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
        ; фаза 1
        movlw   b'00000001'
        movwf   PORTB

        ; затримка (пауза)
        incfsz  0x0C
        goto    $ - 1
        incfsz  0x0D
        goto    $ - 3

        ; фаза 2
        movlw   b'00000010'
        movwf   PORTB

        ; затримка (пауза)
        incfsz  0x0C
        goto    $ - 1
        incfsz  0x0D
        goto    $ - 3

        ; фаза 3
        movlw   b'00000100'
        movwf   PORTB

        ; затримка (пауза)
        incfsz  0x0C
        goto    $ - 1
        incfsz  0x0D
        goto    $ - 3

        ; фаза 4
        movlw   b'00001000'
        movwf   PORTB

        ; затримка (пауза)
        incfsz  0x0C
        goto    $ - 1
        incfsz  0x0D
        goto    $ - 3

        ; фаза 5
        movlw   b'00010000'
        movwf   PORTB

        ; затримка (пауза)
        incfsz  0x0C
        goto    $ - 1
        incfsz  0x0D
        goto    $ - 3

        ; фаза 6
        movlw   b'00100000'
        movwf   PORTB

        ; затримка (пауза)
        incfsz  0x0C
        goto    $ - 1
        incfsz  0x0D
        goto    $ - 3

        ; фаза 7
        movlw   b'01000000'
        movwf   PORTB

        ; затримка (пауза)
        incfsz  0x0C
        goto    $ - 1
        incfsz  0x0D
        goto    $ - 3

        ; фаза 8
        movlw   b'10000000'
        movwf   PORTB

        ; затримка (пауза)
        incfsz  0x0C
        goto    $ - 1
        incfsz  0x0D
        goto    $ - 3

        goto    MainLoop

        END
```

Звісно ж, ні. Такий код є неефективним, особливо якщо потрібно реалізувати великий обсяг команд. Щоб уникнути дублювання, програмісти використовують підпрограми. Підтримка підпрограм реалізована на апаратному рівні мікроконтролерів і мікропроцесорів.

Для роботи з підпрограмами на асемблері існують три команди. Підпрограму створюють із власною міткою, наприклад «Pause». Мітка обов’язкова, оскільки підпрограма повинна мати визначене місце початку, а також надає смислове значення коду. Всередині підпрограми можна виконувати будь-які операції, наприклад паузу.

Виклик підпрограми здійснюється командою call, аргументом якої є мітка підпрограми. Підпрограма повинна завершуватися командою return або retlw. Команда return повертає виконання до інструкції після точки виклику. Команда retlw працює так само, але додатково завантажує в акумулятор указане константне значення, що зручно, коли потрібно повернути результат з підпрограми. Але у типовому випадку застосовують лише стандартну зв’язку call – return, тому зупинимось саме на ній.

Світловий ефект «Вогник, що біжить»

```
        org     0x000

        ; налаштування порту B
        bsf     STATUS, RP0
        movlw   b'00000000'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
        ; фаза 1
        movlw   b'00000001'
        movwf   PORTB
        call    Pause

        ; фаза 2
        movlw   b'00000010'
        movwf   PORTB
        call    Pause

        ; фаза 3
        movlw   b'00000100'
        movwf   PORTB
        call    Pause

        ; фаза 4
        movlw   b'00001000'
        movwf   PORTB
        call    Pause

        ; фаза 5
        movlw   b'00010000'
        movwf   PORTB
        call    Pause

        ; фаза 6
        movlw   b'00100000'
        movwf   PORTB
        call    Pause

        ; фаза 7
        movlw   b'01000000'
        movwf   PORTB
        call    Pause

        ; фаза 8
        movlw   b'10000000'
        movwf   PORTB
        call    Pause

        goto    MainLoop

Pause:  ; затримка (пауза)
        incfsz  0x0C
        goto    Pause
        incfsz  0x0D
        goto    Pause
        return

        END
```

Для повного розуміння того, як працює та, чи інша команда, можна звернутися до технічної документації в якій детально описано роботу кожної з них.

Команди для роботи з підпрограмами

CALL	Call Subroutine
Syntax:	[*label*] CALL k
Operands:	$0 \le k \le 2047$
Operation:	(PC)+ 1→ TOS, k → PC<10:0>, (PCLATH<4:3>) → PC<12:11>
Status Affected:	None
Description:	Call Subroutine. First, return address (PC+1) is pushed onto the stack. The eleven-bit immediate address is loaded into PC bits <10:0>. The upper bits of the PC are loaded from PCLATH. CALL is a two-cycle instruction.

RETURN	Return from Subroutine
Syntax:	[*label*] RETURN
Operands:	None
Operation:	TOS → PC
Status Affected:	None
Description:	Return from subroutine. The stack is POPed and the top of the stack (TOS) is loaded into the program counter. This is a two-cycle instruction.

RETLW	Return with Literal in W
Syntax:	[*label*] RETLW k
Operands:	$0 \le k \le 255$
Operation:	k → (W); TOS → PC
Status Affected:	None
Description:	The W register is loaded with the eight-bit literal 'k'. The program counter is loaded from the top of the stack (the return address). This is a two-cycle instruction.

Підпрограма починає виконуватися після команди **call**

Завершення виконання підпрограми відбувається після команд **return** або **retlw**

Тож, давайте розберемо, як використовувати підпрограму Pause, на прикладі нашої першої практичної роботи «Вітаю, світе!». Програма наведена для того, щоб далі було легше розглядати роботу механізму виклику підпрограм на апаратному рівні. Чим простіший код, тим легше зрозуміти принципи.

У прикладі є основний цикл: увімкнення світлодіода, виклик підпрограми паузи, вимкнення світлодіода, повторний виклик паузи та повернення в початок циклу. Програма елементарна, але добре підходить для демонстрації роботи підпрограм.

На мовах високого рівня підпрограми використовуються постійно: у вигляді бібліотек, фреймворків, власних функцій. Проте багато низькорівневих деталей там не розглядаються. У мікроконтролерах уникнути цих деталей неможливо – потрібно розуміти, як саме працює механізм виклику та повернення з підпрограми.

Програма «Вітаю, світе!»

```
        org     0x000

        ; налаштування порту B
        bsf     STATUS, RP0
        movlw   b'11111110'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
        ; засвітити світлодіод
        bsf     PORTB, RB0
        call    Pause

        ; загасити світлодіод
        bsf     PORTB, RB0
        call    Pause

        goto    MainLoop

Pause:  ; затримка (пауза)
        incfsz  0x0C
        goto    Pause
        incfsz  0x0D
        goto    Pause
        return

        END
```

- При виконанні команди **call** мікроконтролер запам’ятовує адресу, в якій вона розташована
- Наступною командою, що буде виконана після команди **call**, буде команда, адреса якої вказана в цій команді (у прикладі – команда, що має мітку «Pause»)
- Наступною командою, що буде виконана після команди **return** (**retlw**), буде команда, розташована після команди **call**

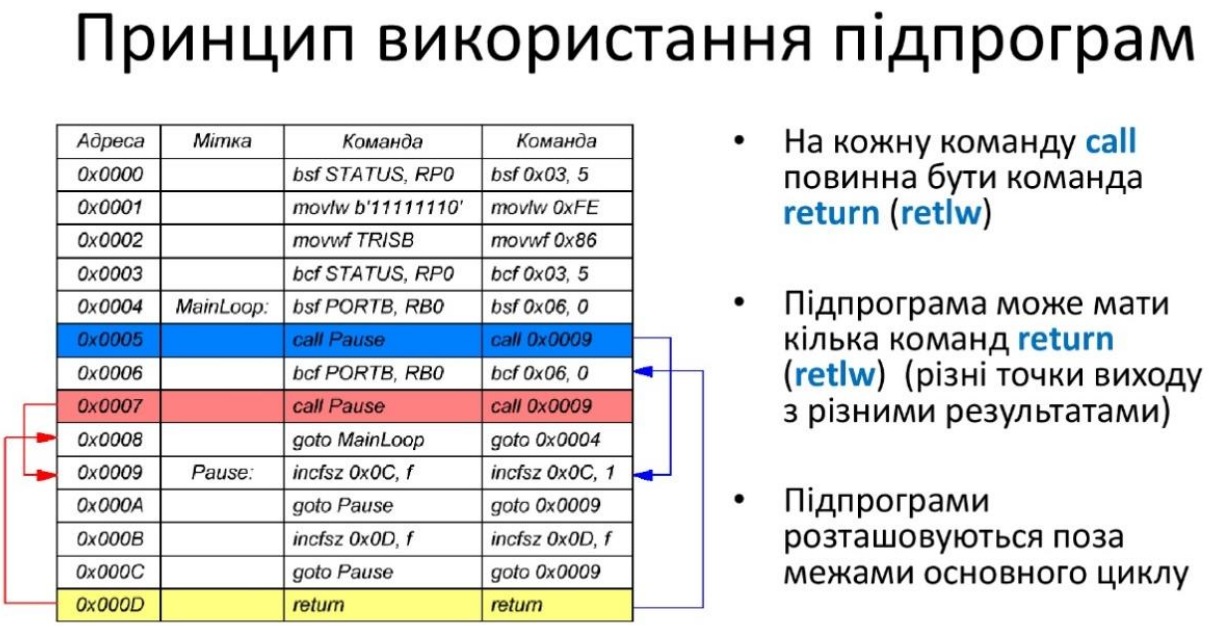
Ось як перша програма розташовується в пам'яті програм. Рядки з адресами та мітками – це те, що ви бачите в редакторі коду, а компілятор у кінцевому результаті перетворює все на числові значення: адреси, номери бітів та інші параметри. Коли ви пишете підпрограму з міткою і викликаєте її командою call, компілятор знаходить інструкцію з цією

міткою і підставляє її реальну адресу. У прикладі підпрограма розташована за адресою 0x0009, і саме ця адреса підставляється замість мітки.

У звичайному режимі ядро мікроконтролера виконує команди послідовно: перша, друга, третя і так далі. Винятком є команди переходу, зокрема goto. Команда call працює аналогічно: виконання переходить на адресу підпрограми. Після виклику ми опиняємось на адресі 0x0009, виконуємо інструкції підпрограми, а коли зустрічається команда return, виконання повертається до інструкції, що йде після тієї, де був викликаний call.

При повторному виклику підпрограми перехід знову здійснюється на адресу 0x0009, а повернення – до наступної команди після другого виклику call. Усе це відбувається автоматично: мікроконтролер самостійно зберігає адресу повернення та відновлює її при виконанні return.

Підпрограма може мати кілька точок виходу, якщо в ній є розгалуження та декілька команд return. Усі вони коректно повертають виконання до відповідної точки виклику. Підпрограми завжди розташовуються поза межами основного циклу програми.

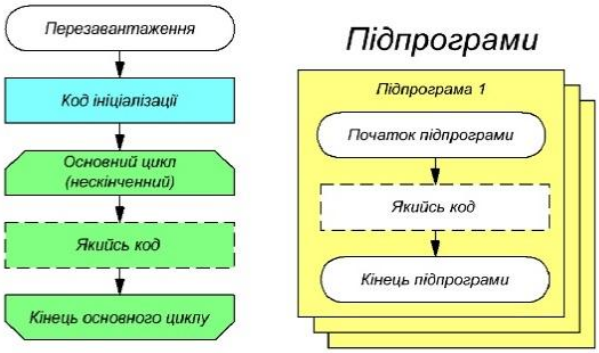


Типова структура програмного забезпечення для мікроконтролера складається з кількох логічних блоків. Перший – перезавантаження, після нього один раз виконується код ініціалізації, у якому налаштовується мікроконтролер для подальшої роботи, далі починається нескінченний основний цикл, у якому виконується основна логіка програми.
Підпрограми можуть бути розташовані як після основного циклу, так і перед ним, але важливо зробити так, щоб вони не виконувалися без виклику call. Наприклад, після ініціалізації можна виконати команду goto main_loop, щоб перескочити через область пам’яті, у якій розташовані підпрограми, і перейти одразу в основний цикл.

Таким чином, код можна структурно поділити на ініціалізацію, основний цикл та підпрограми. У пам’яті програм ці частини розташовані окремими блоками, і важливо, щоб підпрограми не виконувалися автоматично, а запускалися лише через виклик call.

Типовий алгоритм програмного забезпечення мікроконтролера

	Адреса	Мітка	Команда	Команда
Ініціалізація	0x0000		bsf STATUS, RP0	bsf 0x03, 5
	0x0001		movlw b'11111110'	movlw 0xFE
	0x0002		movwf TRISB	movwf 0x86
	0x0003		bcf STATUS, RP0	bcf 0x03, 5
Основний цикл	0x0004	MainLoop:	bsf PORTB, RB0	bsf 0x06, 0
	0x0005		call Pause1	call 0x0009
	0x0006		bcf PORTB, RB0	bcf 0x06, 0
	0x0007		call Pause2	call 0x000E
	0x0008		goto MainLoop	goto 0x0004
Підпрограма Pause1	0x0009	Pause1:	incfsz 0x0C, f	incfsz 0x0C, 1
	0x000A		goto Pause1	goto 0x0009
	0x000B		incfsz 0x0D, f	incfsz 0x0D, f
	0x000C		goto Pause1	goto 0x0009
	0x000D		return	return
Підпрограма Pause2	0x000E	Pause2:	incfsz 0x0C, f	incfsz 0x0C, 1
	0x000F		goto Pause2	goto 0x000E
	0x0010		return	return



У пам'яті програм підпрограми розташовуються поза межами основного циклу

Давайте розглянемо приклад із двома варіантами коду. Код ліворуч працює коректно, тоді як правий варіант на цьому мікроконтролері працювати не буде. У першому варіанті є сім підпрограм. Підпрограма SUB01 викликає SUB02, SUB02 викликає SUB03 і так далі. Виклики підпрограм усередині інших підпрограм є допустимими. Кожна з них завершується командою return, і така структура працює стабільно. Восьма підпрограма також має код і завершується return.

У другому прикладі код виглядає подібно, але всередині восьмої підпрограми викликається дев'ята (SUB09), і повернення відбувається вже з дев'ятої підпрограми. Такий код для даного мікроконтролера коректно працювати не буде. Питання: чому та де мікроконтролер запам'ятовує адреси, куди повертатися? Адже коли виконується команда call, процесор мусить запам'ятати точку, з якої цю підпрограму було викликано.

Особливості використання підпрограм

```
Sub01:
        ; якийсь код
        call    Sub02
        return

Sub02:
        ; якийсь код
        call    Sub03
        return

Sub03:
        ; якийсь код
        call    Sub04
        return

Sub04:
        ; якийсь код
        call    Sub05
        return

Sub05:
        ; якийсь код
        call    Sub06
        return

Sub06:
        ; якийсь код
        call    Sub07
        return

Sub07:
        ; якийсь код
        call    Sub08
        return

Sub08:
        ; якийсь код
        return
```

Цей код буде працювати

```
Sub01:
        ; якийсь код
        call    Sub02
        return

Sub02:
        ; якийсь код
        call    Sub03
        return

Sub03:
        ; якийсь код
        call    Sub04
        return

Sub04:
        ; якийсь код
        call    Sub05
        return

Sub05:
        ; якийсь код
        call    Sub06
        return

Sub06:
        ; якийсь код
        call    Sub07
        return

Sub07:
        ; якийсь код
        call    Sub08
        return

Sub08:
        ; якийсь код
        call    Sub09
        return

Sub09:
        ; якийсь код
        return
```

Цей код працювати не буде

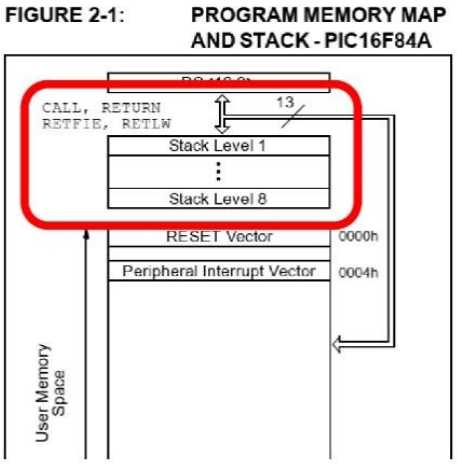
Така область пам'яті називається стек. Він є в будь-якому мікропроцесорі чи мікроконтролері, хоча його реалізація може суттєво відрізнятися залежно від сімейства.

У випадку нашого мікроконтролера PIC1684F, як видно з технічної документації, стек працює доволі специфічно. Програмісту він повністю недоступний: його неможливо прочитати, змінити чи навіть дізнатися, скільки значень там зараз знаходиться. Єдиний спосіб впливати на нього – це виконувати інструкції, які змінюють стек автоматично.

Такими інструкціями є call, return, retlw, а також retfie, що використовується при поверненні з переривання. Крім того, стек оновлюється кожного разу, коли виникає саме переривання.

Головна особливість – обмеження глибини стека. У мікроконтролерах середнього сімейства на кшталт нашого він може зберегти лише вісім адрес повернення. Якщо глибина вкладених викликів перевищує цю межу, стек переповнюється, і повернення з підпрограм починає працювати некоректно. Саме тому код, у якому дев'ята підпрограма викликається всередині восьмої після вже семи попередніх рівнів, перестає працювати правильно.

Стек



- **Стек** – спеціальна область пам'яті, де зберігаються адреси команд з пам'яті програм (інформація про те, куди повертатися)
- Стек знаходиться поза межами адресного простору, тому він недоступний програмісту
- Зміст стеку змінюється при виконанні команд **call**, **return**, **retlw**, **retfie** та при виникненні переривань
- **Стек має 8 комірок пам'яті, тому може запам'ятати лише 8 адрес**
- **При переповненні стеку не виникає жодних помилок в роботі апаратної частини**

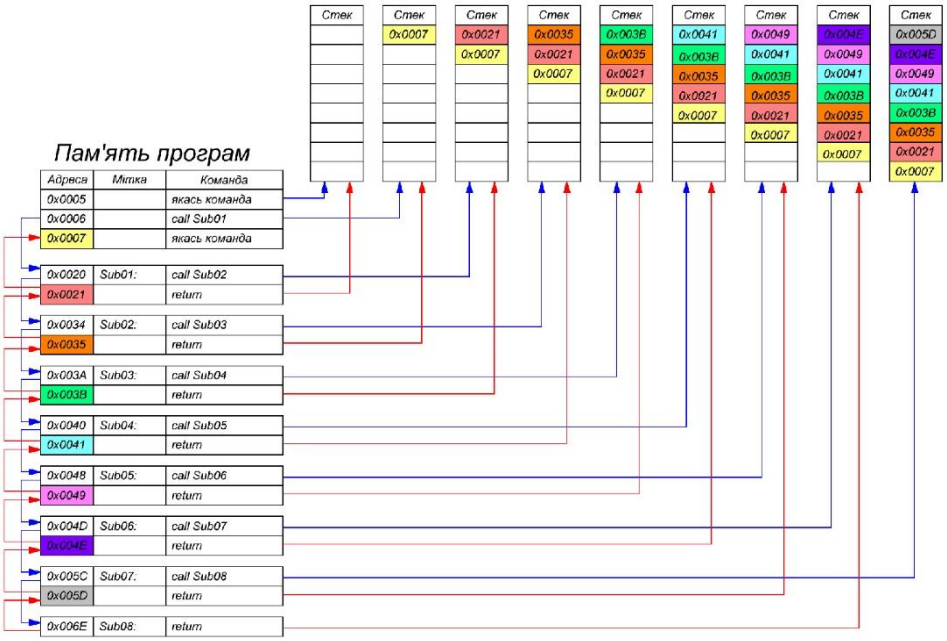
Тепер давайте розберемося, що відбувається всередині програми з урахуванням роботи стеку. У головному циклі виконується звичайний код, а сам стек на цей момент порожній: він має вісім комірок, і хоча там може залишатися якісь дані з попередньої роботи, вони не мають жодного значення для поточного виконання. Як тільки процесор зустрічає команду call, він одразу ж записує у стек адресу повернення. Ця адреса — це адреса самої команди call, збільшена на одиницю, адже після повернення програмі потрібно продовжити виконання вже з наступної інструкції. Наприклад, якщо команда call розташована за адресою 0x0006, то у першу комірку стеку записується 0x0007.

Після цього виконання переходить до мітки підпрограми, наприклад до SUB01. Коли всередині SUB01 знову зустрічається виклик підпрограми, відбувається той самий процес. Адреса повернення записується у верх стеку, а попереднє значення зсувається далі вниз — це робить сам мікроконтролер, програміст до цього не має жодного доступу. Таким чином стек поступово заповнюється, а кожен новий call зсуває попередні дані та займає верхню комірку.

Так процес триває доти, доки викликаються підпрограми SUB02, SUB03 і так далі. Кожного разу нова адреса повернення опиняється у першій комірці стеку, а все, що було там раніше, зміщується вниз. У момент, коли програма доходить до восьмого вкладеного виклику й виконує call SUB08, у стек записується вже восьма адреса повернення — наприклад 5D замість 5C, збільшена на одиницю. Оскільки стек циклічно зміщується вниз, всі попередні значення також переміщаються, і ми бачимо, як вміст стеку постійно «рухається по колу».

Після переходу до SUB08 виконуються її команди, які не впливають на стек, поки не з'явиться інструкція return. У цей момент мікроконтролер бере першу адресу зі стеку й переходить саме до тієї інструкції, яка повинна виконуватися після виклику SUB08. Фактично return робить те саме, що й команда goto на адресу, яку процесор витягнув зі стеку. Після повернення стек зсувається у зворотному напрямку, і той запис, який раніше був другим, стає першим.

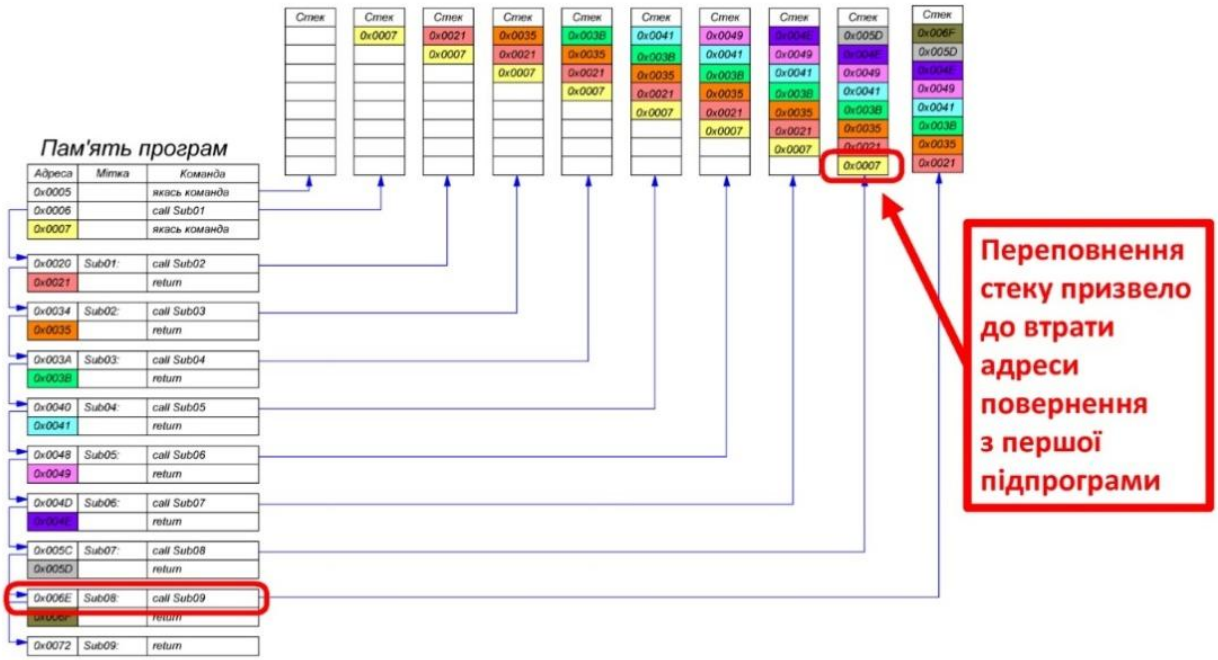
Таким чином програма повертається до місця, де колись була викликана SUB07, продовжує виконання з наступної інструкції і, коли дійде до її return, стек знову підіймається на один рівень вгору. Цей процес буде повторюватися, доки всі вісім вкладених підпрограм не завершаться, повертаючи виконання одна за одною, аж поки стек повністю не відкотиться до початкового стану й програма не повернеться до інструкції з адресою 0x0007 – тієї самої, що й після першого виклику.



Тепер дивимось, як буде працювати правий варіант коду. Він буде поводиться нормально лише доти, поки виклики підпрограм не доходять до дев'ятого рівня. На цьому етапі мікроконтролер знову записує адресу повернення у нульову комірку стеку, зсуваючи всі попередні дані вниз. Саме тут перша адреса – 0x0007, яка відповідала найпершому виклику підпрограми – зникає остаточно. Це і є переповнення стеку: повернутися за цією адресою більше неможливо.

Зовні все ще здається, що програма працює коректно. SUB09 виконується, повернення з попередніх підпрограм також відбуваються, але фактично одна адреса повернення втрачена. Далі все тримається лише на тому, що під час руху назад по вкладених return ми ще маємо коректну адресу 0x0021 у першій комірці стеку – саме туди програма повернеться після відповідного return.

Але після цього повернення стек спорожніє. І наступна команда, яку мікроконтролер зустрічає, знову є return. Виникає питання: куди повертатися, якщо стек порожній?

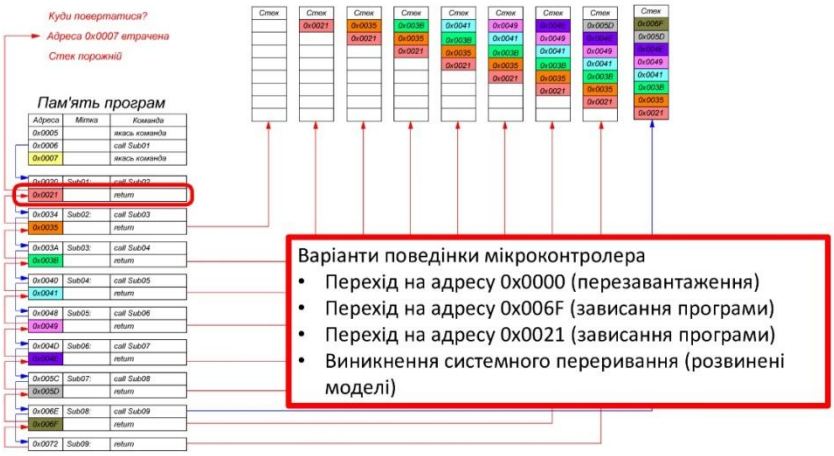


Щоб знайти відповідь на це питання треба занурюватися в глибину і розбиратися, як апаратно працює конкретно ваш стек. Якщо у вас більш розвинений мікроконтролер і він має сучасне ядро, при переповненні стеку виникає системне переривання. Програміст може написати обробник такого переривання і визначити, де саме виникла помилка. Це стосується розвинених мікроконтролерів, але не PIC16F84.

У PIC16F84 при виконанні команди return за порожнього стеку мікроконтролер перезавантажується, оскільки наступною командою стає перехід на адресу 0, що відповідає рестарту програми. У результаті програма може зависнути та виконуватись у замкненій ділянці коду, не повертаючись у нормальну частину.

Такі поведінкові особливості є недокументованими та виявляються експериментально. Навіть у межах одного сімейства різні моделі можуть поводитися по-різному. Переповнення стеку можливе і в мовах високого рівня, оскільки стек використовується для збереження адрес і передачі змінних у підпрограми. У мікроконтролерах із малим стеком, до якого немає доступу, такий механізм застосувати неможливо.

При використанні підпрограм потрібно враховувати, що вкладені виклики, характерні для мов високого рівня та ООП, у мікроконтролерах з обмеженим стеком можуть призвести до переповнення, особливо при рекурсії, коли підпрограма викликає саму себе.



Такий підводний камінь виникає, якщо не контролювати стан стеку. Це дуже підступні помилки, які важко відстежити. Особливо складно, коли використовуються переривання, оскільки в них також можуть виконуватися підпрограми.

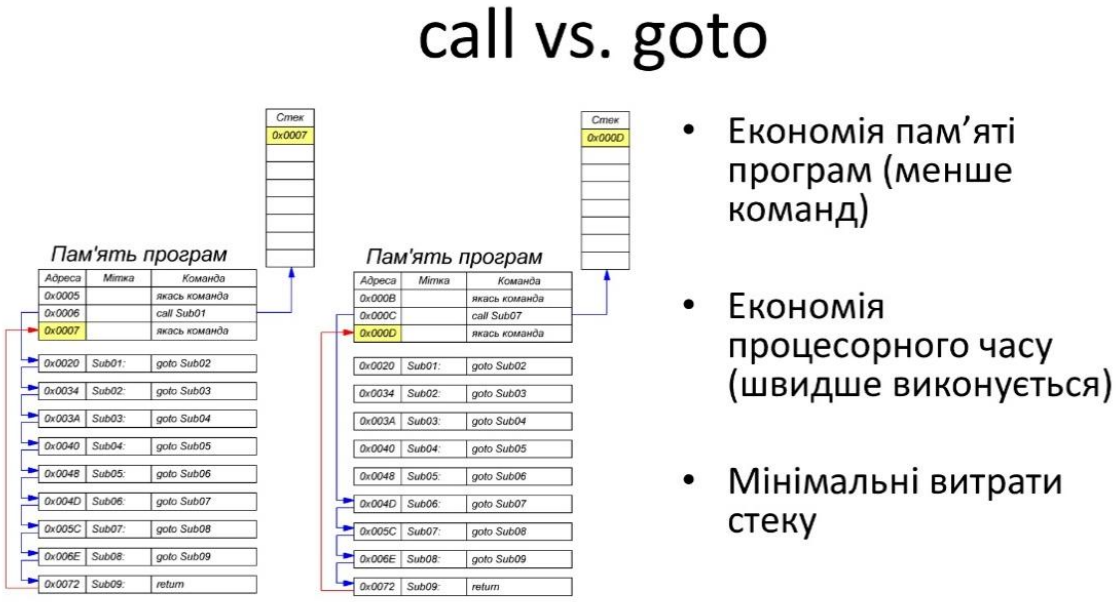
Може статися, що програма працює коректно в більшості випадків, але один раз із тисячі виникає переповнення стеку. Це означає, що була допущена ситуація з більш ніж вісьмома вкладеними викликами, і система переходить у некоректний режим виконання.

Також існує прийом, який дозволяє зменшити навантаження на стек. Його суть полягає в тому, що замість використання команди call застосовується goto. При такому підході з коду прибираються команди return. Фактично підпрограма не має явного завершення, а замість виклику наступної підпрограми виконується звичайний перехід до неї.

Коли використовується вкладення підпрограм, перехід відбувається послідовно від однієї підпрограми до іншої за допомогою команд переходу, аж до останньої. Повернення виконується лише один раз, оскільки в стеці зберігається адреса першого виклику команди call, і цього достатньо для повернення керування.

У класичному варіанті кожен виклик підпрограми зберігає адресу повернення в стеці, тому при глибокому вкладенні накопичується велика кількість записів. Після виконання останньої підпрограми відбувається послідовне повернення через всі вкладені виклики. У запропонованому підході можна викликати будь-яку підпрограму однією командою call, після чого виконати переходи до наступних підпрограм, а наприкінці повернутися однією командою return.

Такий підхід дозволяє зменшити кількість команд return у програмі, скоротити використання пам’яті програм і процесорного часу, а також мінімізувати використання стеку. Прийом застосовується в асемблері і працює не лише на мікроконтролерах, а й на будь-яких мікропроцесорах.



Єдине зауваження полягає в тому, що на мовах високого рівня використання команди безумовного переходу не рекомендується. Якщо програміст у мовах C, Pascal або Python використовує такі переходи, це свідчить про неправильну побудову алгоритму та невміння коректно структурувати програму.

Об’єктно-орієнтоване програмування взагалі суперечить такому підходу, оскільки його парадигма базується на розділенні програми на окремі об’єкти. У цьому випадку один і той самий код може бути безпосередньою частиною різних об’єктів, а не просто вкладеним фрагментом, тому реалізувати подібні переходи в межах об’єктно-орієнтованої моделі неможливо.

На мовах високого рівня такі переходи формально можливі, але забороняються або не рекомендуються в навчальній і практичній літературі. Натомість асемблер нормально реагує на використання безумовних переходів, і без команди goto написання програм на асемблері є неможливим.

Програмування на асемблері передбачає ручне продумування та реалізацію оптимізацій без покладання на автоматичні оптимізатори, які не здатні забезпечити максимальну ефективність. Лише на асемблері можна створити компактний код, що виконується максимально швидко. Саме тому драйвери та низькорівневі функції часто містять асемблерні вставки, особливо в тих частинах, які виконуються дуже часто і потребують значних обчислювальних витрат.

Мови високого рівня є більш витратними з точки зору використання процесорного часу та швидкодії, тому для критичних ділянок коду асемблер залишається єдиним ефективним варіантом. Для цього необхідно добре розуміти роботу стеку та механізми переходів call і goto.

Повертаємося до світлофора. Згідно з технічним завданням будуть різні затримки: червоне світло може горіти довго, жовте – менше, зелене може не горіти або блимати. Для цього рекомендується зробити дві підпрограми. Перша формує базову затримку, основну опору, наприклад еквівалентну 0,1 секунді; це підпрограма basedelay.

У базовій затримці використовуються дві комірки пам’яті 0x0C і 0x0D. Раніше їх збільшували, але тут доцільніше зменшувати на одиницю; принцип той самий: коли з нуля зменшити на одиницю, отримуємо 255. У цьому прикладі inc і dec працюють однаково за ефектом циклу, відрізняється лише напрям зміни: inc збільшує, dec зменшує. Перед виконанням циклу ці комірки потрібно ініціалізувати, записавши туди значення, інакше перша затримка буде відрізнятися від наступних. Це проявляється як те, що перше блимання світлодіода за часом відрізняється від усіх інших, бо на старті в 0x0C і 0x 0D містяться довільні значення, а при перезавантаженні мікроконтролера комірки не скидаються в нуль.

Після запису початкових значень виконується цикл, який повторюється доти, доки значення в цих комірках не стане рівним нулю, після чого відбувається повернення з підпрограми. При виборі початкових чисел потрібно враховувати тактову частоту: у світлофорі вона менша, тому ті самі значення дадуть значно більшу затримку. Якщо робити, як раніше, багаторазовий прохід із заповненням до 256, затримка може вийти близько півхвилини саме через меншу тактову частоту, тому доцільніше записувати не максимальні значення, а менші, наприклад на кшталт 128, хоча і це може бути забагато.

Друга підпрограма задає користувацьку затримку як кратну базовій. Її ідея в тому, що вводиться ще одна змінна за окремою адресою, куди завантажується число, яке показує, скільки разів потрібно викликати базову затримку. Це число завантажується в акумулятор перед викликом, далі з акумулятора записується в комірку 0x0E, після чого викликається delay, потім значення в 0x0E зменшується на одиницю і виконується перехід назад у цикл, доки не стане нулем. Для цього потрібна мітка перед повторним викликом; у фрагменті її не вистачає, тому її слід поставити перед викликом або замінити на відносний перехід на кшталт поточної адреси мінус два. Так формується потрібна затримка як задана кількість повторів базової підпрограми.

```
UserDelay:
        ; записуємо початкове значення
        ; в комірку з адресою 0x0E
        movfw   0x0E

BaseLoop:
        call    BaseDelay

        decfsz  0x0E, f
        goto    BaseLoop

        return

BaseDelay:
        ; записуємо початкове значення
        ; в комірку з адресою 0x0D
        movlw   d'128'
        movwf   0x0D

        ; записуємо початкове значення
        ; в комірку з адресою 0x0C
        movlw   d'0'
        movwf   0x0C

DelayLoop:
        ; зменшуємо числа в комірках
        ; з адресами допоки вони
        ; не стануть рівними 0
        decfsz  0x0C, f
        goto    DelayLoop
        decfsz  0x0D, f
        goto    DelayLoop

        return
```

Формування довільної затримки

- **Підпрограма BaseDelay** формує базову затримку (наприклад 0,1 с), тривалість якої визначається числом, що завантажу'ється в комірку пам'яті з адресою 0x0D перед переходом в основний цикл затримки (DelayLoop)
- **Підпрограма UserDelay** формує довільну затримку, яка визначається числом в акумуляторі, яке зберігається в комірці пам'яті з адресою 0x0E та визначає скільки разів буде викликана підпрограма BaseDelay

Підпрограма UserDelay фактично є підпрограмою з параметрами (на мові асемблеру мікроконтролерів PIC середнього сімейства таким чином можна передати тільки один параметр)

Розглянемо використання. Якщо потрібна лише базова затримка, виконується прямий виклик call basedelay, після чого відбувається повернення і програма продовжує виконання.

Якщо потрібна кратна затримка, спочатку в акумулятор записується кількість повторів, після чого викликається підпрограма userdelay. Вона формує затримку шляхом багаторазового виконання базової затримки. Значення в акумуляторі визначає, скільки разів буде виконано basedelay.

Максимальна кількість повторів становить 256 разів. Це досягається при записі нуля, оскільки максимальне значення лічильника дорівнює 255, а нуль відповідає повному циклу з 256 проходів. Якщо, наприклад, базова затримка дорівнює одній секунді, то максимальна сформована затримка становитиме 25,6 секунди.

Якщо потрібна ще більша затримка, підпрограму userdelay викликають кілька разів послідовно.

```
UserDelay:
        ; записуємо початкове значення
        ; в комірку з адресою 0x0E
        movfw   0x0E

        call    BaseDelay

        decfsz  0x0E, f
        goto    BaseLoop

        return

BaseDelay:
        ; записуємо початкове значення
        ; в комірку з адресою 0x0D
        movlw   d'128'
        movwf   0x0D

        ; записуємо початкове значення
        ; в комірку з адресою 0x0C
        movlw   d'0'
        movwf   0x0C

DelayLoop:
        ; зменшуємо числа в комірках
        ; з адресами допоки вони
        ; не стануть рівними 0
        decfsz  0x0C, f
        goto    DelayLoop
        decfsz  0x0D, f
        goto    DelayLoop

        return
```

Формування затримок

За допомогою підпрограм BaseDelay та UserDelay можна сформувати затримку

1 * BaseDelay … 256 * BaseDelay

```
; формуємо затримку 25,6 с
; 256 * 0,1 = 25,6 с
movlw   d'0'
call    UserDelay

; формуємо затримку 60 с
movlw   d'200'
call    UserDelay
movlw   d'200'
call    UserDelay
movlw   d'200'
call    UserDelay
```

```
; формуємо базову
; затримку BaseDelay
; BaseDelay = 0,1 с
call    BaseDelay

; ---

; формуємо затримку 0,1 с
; 1 * 0,1 = 0,1 с
movlw   d'1'
call    UserDelay

; ---

; формуємо затримку 0,2 с
; 2 * 0,1 = 0,2 с
movlw   d'2'
call    UserDelay

; ---

; формуємо затримку 2 с
; 20 * 0,1 = 2 с
movlw   d'20'
call    UserDelay

; ---

; формуємо затримку 15 с
; 150 * 0,1 = 15 с
movlw   d'150'
call    UserDelay
```

У підсумку постає питання, як усі команди переходів і затримки прив'язати до реального часу. Це питання потребує розуміння роботи тактових генераторів, яке буде розглянуте на наступній лекції, оскільки обсяг інформації для одного заняття занадто великий.

На цьому етапі необхідно розуміти, що всі команди асемблера виконуються за певний проміжок часу. Кожна команда відповідає визначеній кількості тактових імпульсів, тому швидкість виконання програми безпосередньо залежить від частоти тактового генератора.

До цього моменту використовувалася тактова частота 4 МГц, тоді як у світлофорі застосовується частота 32768 Гц, так званий часовий кварц. Саме різниця частот визначає зміну часу виконання тих самих програмних конструкцій.

Частота виконання залежить не лише від тактового генератора, а й від архітектури мікроконтролера. В різних мікроконтролерах одна й та сама команда може виконуватися за різну кількість тактів. Наприклад, у PIC16F84 стандартна команда виконується за чотири такти, в інших мікроконтролерах – за один або за шістнадцять тактів. Тому сама по собі тактова частота не повністю визначає швидкість виконання асемблерного коду.

Прив’язка до реального часу

```
DelayLoop:
        ; зменшуємо числа в комірках
        ; з адресами допоки вони
        ; не стануть рівними 0
        decfsz  0x0C, f
        goto    DelayLoop
        decfsz  0x0D, f
        goto    DelayLoop
```

Скільки часу виконується цей код?

Що впливає на час виконання програмного коду

- Апаратна основа (який саме мікроконтролер)
- Частота роботи тактового генератора
- Призначення команди, яка виконується (різні команди потребують різного часу виконання

Різні команди можуть потребувати різної кількості тактів для виконання, і це безпосередньо впливає на час роботи програми. Поки що ці часові параметри пропонується визначати експериментально до наступної практичної роботи. Далі ми детально розглянемо роботу тактових генераторів, внутрішнє виконання команд, причини різної кількості тактів та способи їх налаштування в програмі.

Висновки

- Використання підпрограм зменшує витрати пам’яті програм
- При використанні підпрограм використовується стек, який є слабким місцем будь-якого програмного забезпечення
- При переповненні стеку програмне забезпечення веде себе непедбачувано

- На мовах асемблеру існують методи програмування, що дозволяють додатково зменшити витрати пам’яті програм, оперативної пам’яті та стеку, але їх не рекомендується використовувати на мовах високого рівня, і не можна використовувати при об’єктно-орієнтованому програмуванні

ФОРМУВАННЯ ЧАСОВИХ ІНТЕРВАЛІВ

Сьогодні ми продовжуємо вивчення дисципліни програмування мікроконтролерів та пристроїв Інтернету. Лекція присвячена завершенню великої теми, що стосується формування часових інтервалів. Розглядається прив'язка віртуального часу, у якому функціонує програмне забезпечення, до реального астрономічного часу, в межах якого працюють застосунки, створені на базі мікроконтролера. Основний акцент робиться на зв'язуванні програмного середовища з реальним фізичним світом, у якому безпосередньо функціонує мікроконтролер.

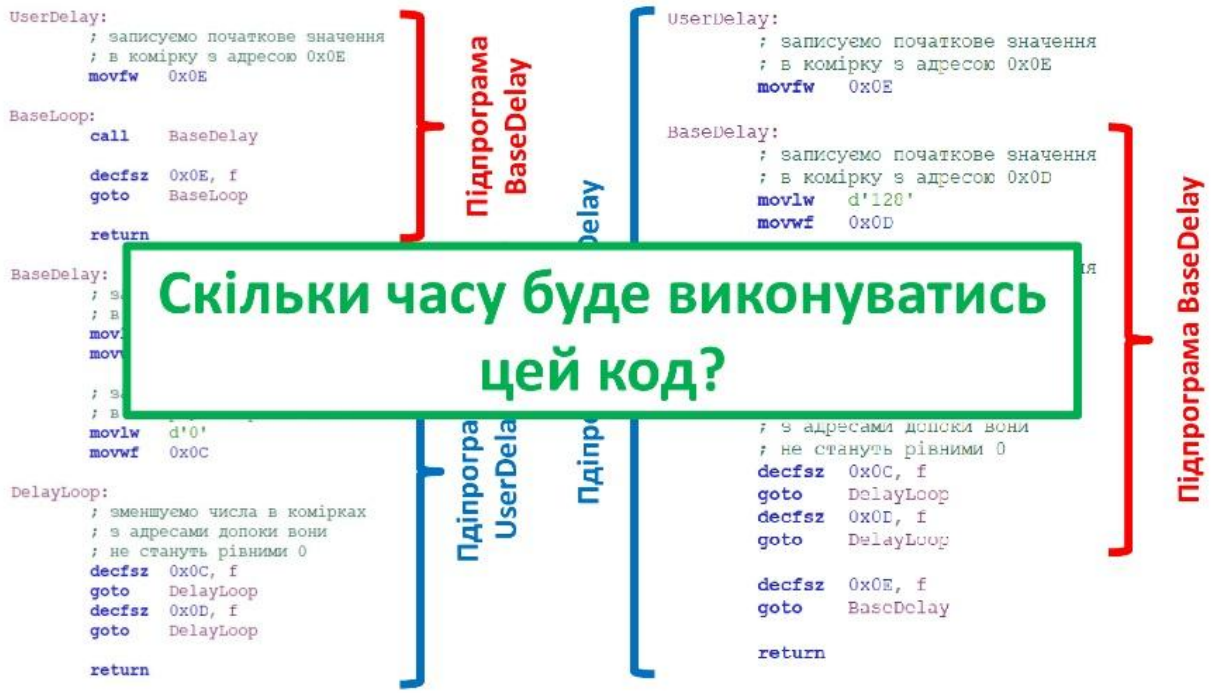
Почнемо поступово розбиратися. Остання практична робота була присвячена програмуванню світлофора. У технічному завданні були задані конкретні часові параметри: тривалість світіння червоного, жовтого та інших сигналів.

На практиці ці значення підбиралися емпірично, тобто «методом наукового тику». Код змінювали, перевіряли результат, оцінювали швидкість роботи, після чого коригували затримки, роблячи їх більшими або меншими, доки робота світлофора не здавалася прийнятною.

На попередній лекції розглядалися підпрограми та був запропонований рекомендований код для формування затримок. Було наведено два варіанти реалізації, які з точки зору кінцевого застосування є еквівалентними. Один варіант простіший для розуміння, але використовує більше ресурсів, інший складніший, проте має меншу кількість команд і коротший за виконанням.

Вибір конкретної реалізації не є принциповим. Кожен програміст формує власний стиль програмування, який з часом стає впізнаваним і унікальним.

Водночас залишилося головне питання, на яке не було отримано відповіді під час виконання попередніх робіт. У всіх застосунках використовувався код, що працює майже в реальному часі, але не аналізувалося, скільки саме часу займає його виконання. Тепер необхідно перейти до пошуку конкретних відповідей щодо часу виконання програм.



Перш за все необхідно з'ясувати роль тактового генератора. Мікроконтролер, мікропроцесор і будь-яка комп'ютерна або цифрова техніка містять вузол, який називається тактовим генератором. Саме він є джерелом роботи всієї системи: якщо тактовий генератор не працює, система не функціонує.

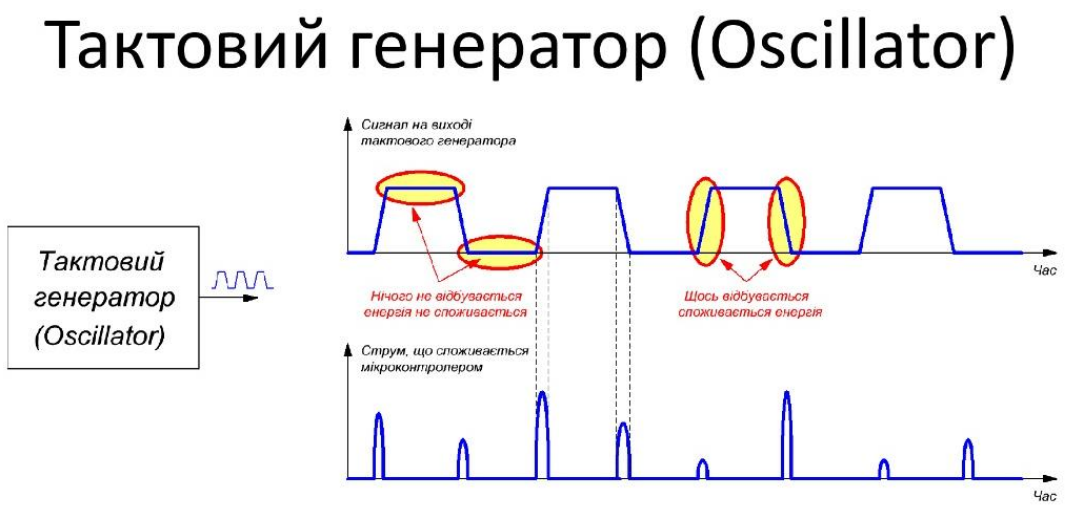
У сучасній цифровій техніці, побудованій на логічних елементах, усі процеси відбуваються лише в моменти зміни сигналу на виході тактового генератора. Тактовий генератор не має входів і самостійно формує послідовність нулів і одиниць з певним часовим інтервалом. Зазвичай коефіцієнт заповнення імпульсів сигналу становить близько 50%, тобто тривалість логічного нуля приблизно дорівнює тривалості логічної одиниці.

Коли сигнал тактового генератора не змінюється і перебуває в стані нуля або одиниці, усередині мікроконтролера чи мікропроцесора не відбувається жодних процесів. Внутрішні вузли при цьому практично не споживають енергію. Струм може протікати лише в зовнішніх колах, наприклад на портах введення-виведення, підключених до фізичної схеми, але внутрішні процеси зупинені.

Якщо розглядати споживання струму мікроконтролером у часі, воно має вигляд коротких імпульсів, які виникають саме в моменти зміни тактового сигналу. Якщо зупинити тактовий генератор, система повністю завмирає і перестає реагувати. При відновленні роботи генератора навіть через тривалий час система продовжить виконання з того самого стану, в якому була зупинена.

Імпульси струму мають різну амплітуду, оскільки кожна зміна тактового сигналу призводить до перемикання різної кількості транзисторів усередині мікропроцесора. Величина струму залежить від того, скільки елементів змінили свій стан під час переходу сигналу. Коли на виході тактового генератора стабільний нуль або одиниця, енергоспоживання мінімальне.

Це важливо враховувати як при розробці апаратної частини, так і при написанні програмного забезпечення, оскільки якість програмного коду безпосередньо впливає на енергоспоживання системи.



Усі процеси в мікроконтролері відбуваються при зміні сигналу на виході тактового генератора

Оскільки вже розглядається робота тактового генератора, доцільно окремо зупинитися на питанні енергоспоживання мікроконтролера. Якщо розглядати процеси на глибшому рівні, нижче рівня електроніки, на рівні фізики виготовлення елементів, базовим елементом усіх цифрових схем є польовий транзистор. Він формується в кристалі шляхом хімічного травлення ділянок з різними фізичними властивостями та нанесення електродів і діелектриків у межах визначеного технологічного процесу.

Транзистор має певні геометричні розміри, а його виводи, хоча й електрично ізольовані, утворюють паразитні електричні ємності. Ці ємності є невід’ємною властивістю структури й не можуть бути усунуті. Саме на перезаряд цих паразитних ємностей витрачається основна частина енергії мікроконтролера. Енергоспоживання ядра не пов’язане

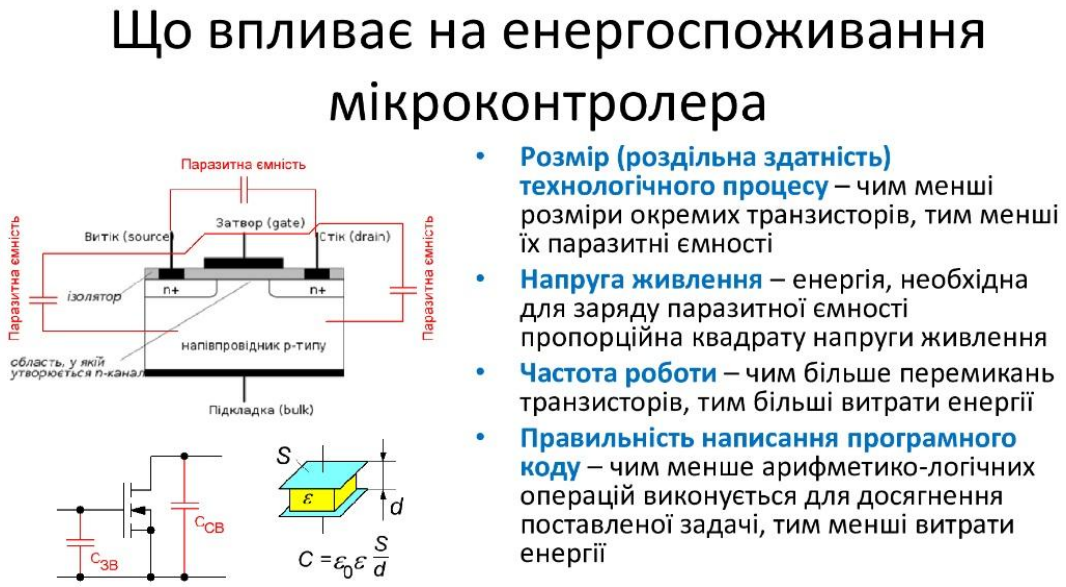
безпосередньо з виконанням арифметико-логічних операцій, а визначається процесами перезаряду внутрішніх ємностей.

Величина енергоспоживання залежить від розмірів транзисторів. Чим менші транзистори, тим менші паразитні ємності і, відповідно, менше енергоспоживання при виконанні тієї самої задачі. Тому енергоспоживання мікроконтролерів і комп'ютерних систем безпосередньо пов'язане з розміром технологічного процесу та можливостями виготовлення малих елементів.

Ще одним важливим параметром є напруга живлення. Для заряджання паразитної ємності необхідна енергія, яка залежить від квадрату напруги. При зменшенні напруги живлення вдвічі енергоспоживання зменшується приблизно в чотири рази за умови виконання тієї самої програми. Також енергоспоживання залежить від частоти роботи, тобто від частоти тактового генератора. Чим вища частота, тим більше перемикань і тим більше споживання енергії.

Саме тому в сучасних мікроконтролерах, особливо тих, що працюють від батарей, основними тенденціями є зменшення напруги живлення, зменшення розміру технологічного процесу та зниження робочої частоти. У технічній документації енергоспоживання часто подається у відносних величинах, наприклад у міліамперах на мегагерц. Це означає, що при збільшенні тактової частоти пропорційно зростає споживаний струм.

Окремо слід враховувати вплив програмного коду. Неоптимальний код із зайвими операціями призводить до додаткових перемикань і, відповідно, до зростання енергоспоживання. Тому при створенні застосунків для мікроконтролерів важливо розуміти, що навіть за умови роботи лише з програмним забезпеченням від якості та оптимальності коду безпосередньо залежить рівень енергоспоживання системи.



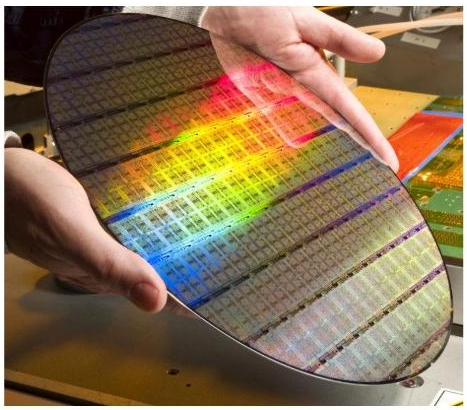
Тепер, давайте подивимось, як виглядають кристали мікросхем. Зовні вони однакові, незалежно від того, чи це мікроконтролери, чи інші інтегральні схеми. Процес виготовлення полягає не в створенні окремих транзисторів або окремих мікроконтролерів. Береться пластина кристалу, на якій одночасно виготовляються десятки, сотні або тисячі мікросхем. Після завершення технологічного процесу пластина розрізається на окремі кристали, які розміщуються в корпусах, після чого до них під'єднуються виводи.

Розвиток технологічних процесів характеризується зменшенням роздільної здатності. Починаючи приблизно з 1971 року, коли з'явилися перші мікросхеми, мінімальний технологічний крок становив близько 10 мкм. Це не безпосередній розмір транзистора, а мінімальна відстань, з якою можуть формуватися елементи, при цьому сам транзистор має дещо більші розміри.

З часом розміри елементів суттєво зменшилися. До 2029-го року компанія Intel, планує перехід до технологічних норм порядку 1,4 нм, тобто зменшення розмірів буде приблизно у десять тисяч разів порівняно з початковими значеннями.

Водночас не всі компанії здатні освоїти найсучасніші технологічні процеси, оскільки створення обладнання з такою точністю є надзвичайно складним. За таких масштабів технології наближаються до розмірів окремих атомів. Імовірно, ще протягом кількох десятків років розвиток відбуватиметься в цьому напрямку, після чого подальше зменшення розмірів транзисторів стане фізично неможливим.

Роздільна здатність технологічних процесів виготовлення напівпровідникових мікросхем



- **10 мкм** – Intel 4004 (1971 рік)
- **3 мкм** – Zilog Z80 (1975 рік), Intel 8086 (1979 рік)
- **1 мкм** – Intel 80386 (1985 рік)
- **0,6 мкм** – Intel Pentium (1994 рік)
- **0,13 мкм** – AMD Athlon XP, Intel Pentium III (початок 2000-х років)
- **45 нм** – Intel Core 2 Quad, Fujitsu SPARC64 VIIIfx (2007 рік)
- **14 нм** – Celeron N3000, N3050, N315, Apple A10 (2015 рік)
- **10 нм** – Apple A11, Snapdragon 845 (2017 рік)
- **7 нм** – Apple A12, Kirin 980, Snapdragon 855, Intel Ice Lake (2019 рік)
- **3 нм** – Cadence Design Systems (перші дослідні зразки) (2018 рік), Samsung (планується у 2021 році)
- **1,4 нм** – Intel (планується у 2029 році)

Розглянемо електронну частину. Струм, який споживає будь-яка цифрова мікросхема, не лише мікроконтролер, а й мікропроцесори та інші цифрові пристрої, має імпульсний характер у вигляді коротких піків. Жодне джерело живлення не здатне безпосередньо забезпечити такі імпульси струму, оскільки воно розташоване на відстані від мікросхеми і з’єднане з нею провідниками, які мають паразитну індуктивність.

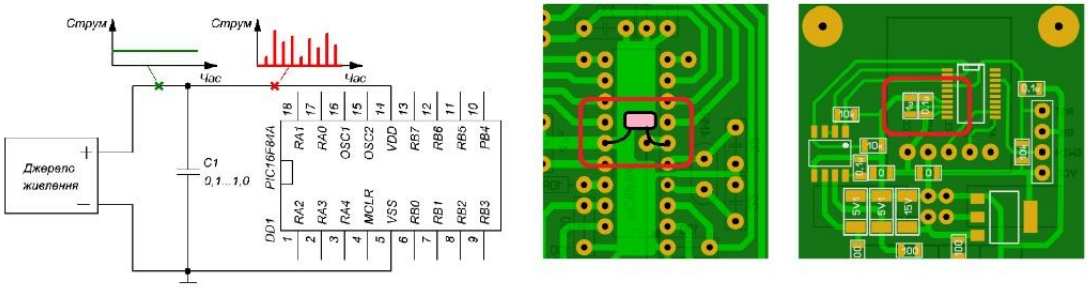
Якщо не вживати додаткових заходів і підключити мікросхему напряму до джерела живлення, вона може працювати некоректно або взагалі не працювати. Як приклад, розглядався випадок із лічильником на двох цифрових мікросхемах без мікроконтролера, який мав відображати числа по колу від 0 до 9. Фактично лічильник показував послідовність 0, 1, 3, 5, 6, 7, 9, тобто два значення випадали. Налаштувань схема не мала, заміна мікросхем не дала результату, і причина несправності тривалий час була незрозумілою. Проблема зникла лише після встановлення конденсатора паралельно до живлення мікросхеми, після чого схема почала працювати коректно.

Причина такої поведінки полягає в неможливості швидко передати імпульсні струми через індуктивність провідників живлення. Для компенсації цього ефекту в колах живлення застосовуються локальні накопичувачі енергії.

У колі живлення будь-якого мікроконтролера обов’язково має бути встановлений малий керамічний конденсатор ємністю приблизно від 10 нФ до 1 мкФ, можливі проміжні значення, наприклад 47 нФ. Цей конденсатор підключається паралельно до виводів живлення мікросхеми і забезпечує локальний запас енергії для імпульсних струмів.

Конденсатор повинен розташовуватися безпосередньо біля виводів живлення мікросхеми і не може бути винесений далеко на платі. Саме тому в мікроконтролерах і мікропроцесорах виводи живлення зазвичай розміщують парами і максимально близько один до одного, щоб між ними було зручно встановити невеликий SMD керамічний конденсатор. Навіть якщо мікроконтролер має кілька пар виводів живлення, вони завжди виконуються парними і розташовуються таким чином, щоб забезпечити можливість встановлення конденсатора безпосередньо біля цих виводів.

Використання блокувальних конденсаторів



Для стабільної роботи мікросхем КМОН-логіки необхідно встановлювати в колах її живлення керамічний конденсатор ємністю 0,047…1,0 мкФ, який усуває коливання напруги в колах живлення, спричинений імпульсним характером споживаного струму

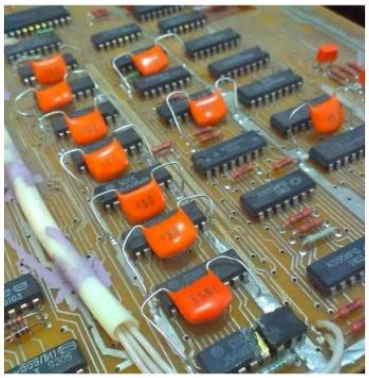
Конденсатор повинен розташовуватись на платі безпосередньо біля мікросхеми

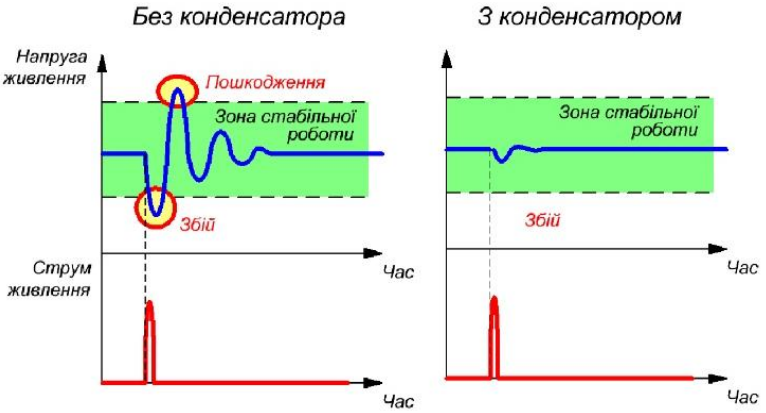
Це є критично важливий момент. Якщо ним знехтувати, виникає ситуація, коли без конденсатора проявляються паразитні параметри провідників живлення і починаються коливальні процеси. У результаті напруга може виходити за межі зони стабільної роботи мікросхеми. Це може призвести до порушення роботи оперативної пам’яті, пошкодження або спотворення її вмісту та інших некоректних ефектів.

У такій ситуації помилку часто намагаються знайти в програмному коді, хоча реальна причина полягає лише у відсутності конденсатора в колі живлення. У найгіршому випадку можливе фізичне пошкодження мікросхеми, хоча це трапляється рідко. Найчастіше проблема проявляється у нестабільній роботі.

Саме тому навіть на старих платах конденсатори часто припаювали безпосередньо до виводів живлення мікросхем. Хоча виводи живлення можуть бути запаралелені, біля кожної мікросхеми встановлюється власний конденсатор, який забезпечує стабільну роботу пристрою.

Використання блокувальних конденсаторів





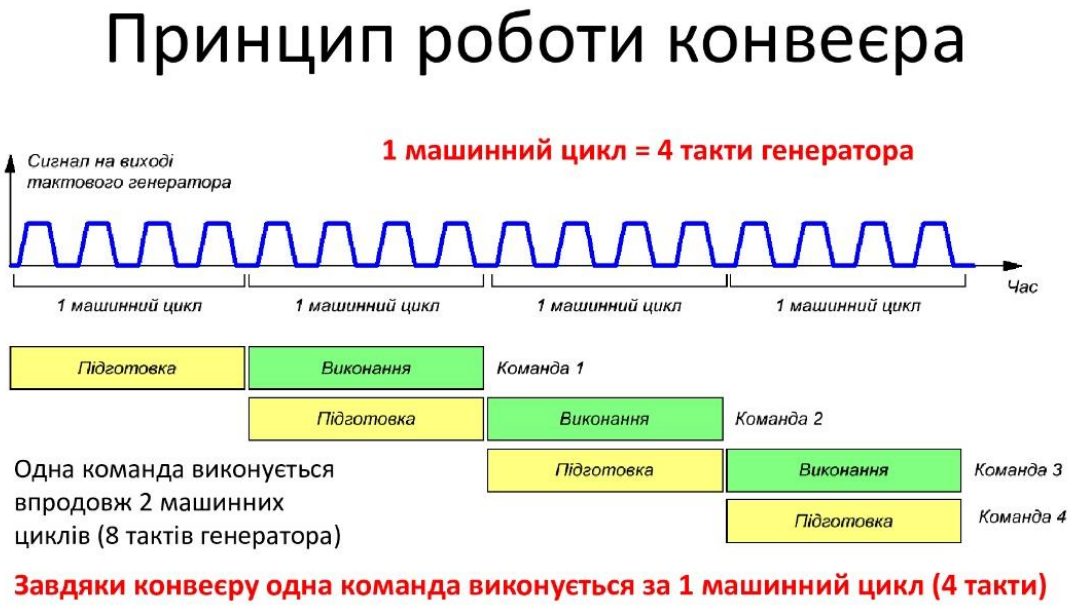
Електронну частину розглянули, тепер переходимо до принципів роботи мікроконтролера. Усі процеси в ньому відбуваються під час зміни тактового сигналу. Виникає питання, скільки тактових імпульсів необхідно для виконання однієї команди, і це значення залежить від конкретного мікроконтролера.

У мікроконтролерах PIC середнього сімейства виконання однієї команди потребує восьми тактових імпульсів. Внутрішньо цей процес умовно поділяється на дві частини. Перші чотири такти відводяться на підготовку: команда зчитується з пам'яті програм, завантажується у внутрішні регістри, можуть підвантажуватися дані з оперативної пам'яті. На цьому етапі стан мікроконтролера не змінюється. Другі чотири такти — це безпосереднє виконання команди, яке вже змінює стан мікроконтролера. Ці чотири такти називаються машинним циклом.

Таким чином, робота мікроконтролера поділяється на машинні цикли, кожен з яких триває чотири тактові імпульси. Одна команда виконується за два машинні цикли, тобто за вісім тактів.

Для підвищення швидкодії розробники реалізували конвеєрне виконання команд. Конвеєр означає, що дві команди обробляються одночасно зі зсувом у часі. Поки одна команда виконується, відбувається підготовка до виконання наступної. Мікроконтролер автоматично зчитує наступну команду з пам'яті програм і готує її до виконання паралельно з виконанням поточної.

Завдяки конвеєру створюється ефект, що команди виконуються вдвічі швидше. У результаті здається, що на виконання однієї команди витрачається не вісім тактових імпульсів і не два машинні цикли, а один машинний цикл, тобто чотири тактові періоди. Саме це значення використовується як базове при подальших розрахунках часу виконання програм.



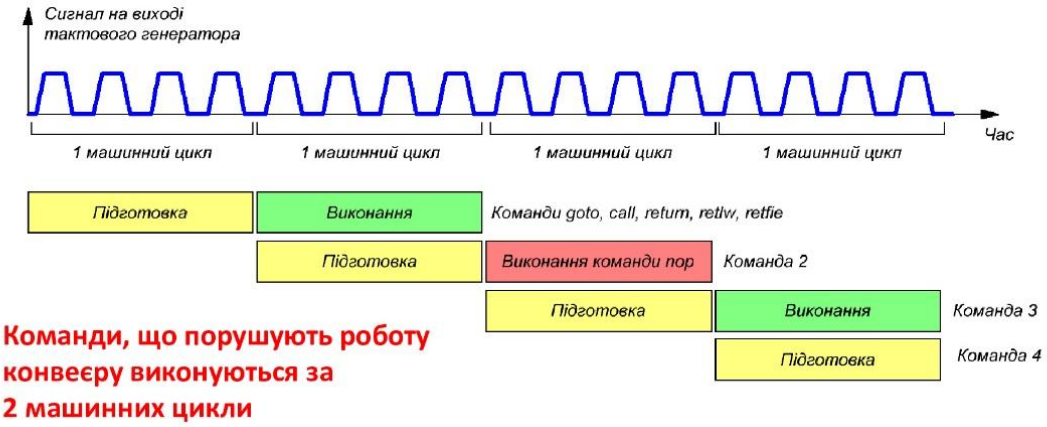
Однак у наборі інструкцій є команди, які порушують роботу конвеєра. До них належать команди, що змінюють природний послідовний порядок виконання команд у пам'яті програм, зокрема goto, call, return, а також команди роботи з перериваннями.

Під час виконання, наприклад, команди goto наступною має виконуватися не команда, яка була заздалегідь підготовлена конвеєром, а команда за адресою, вказаною в цій інструкції. У цей момент підготовка наступної послідовної команди стає некоректною, і робота конвеєра збивається.

Усередині мікроконтролера це вирішується так: замість виконання вже підготовленої команди виконується команда NOP, яка не виконує жодної операції і слугує для вирівнювання роботи конвеєра. Таким чином відбувається втрата одного машинного циклу.

У результаті команди, що порушують роботу конвеєра, такі як goto, call, return і команди обробки переривань, виконуються за два машинні цикли, оскільки їх виконання потребує відновлення коректної роботи конвеєра.

Порушення роботи конвеєру



Таким чином, існують асемблерні команди, які виконуються за один машинний цикл, і команди, виконання яких потребує двох машинних циклів.

Команда NOP наведена як приклад. Вона не виконує жодних дій і не змінює стан мікроконтролера, являючи собою порожню операцію. Попри це, інколи команда NOP використовується в програмному коді, і далі буде показано приклад її застосування.

Команда nop

NOP	No Operation
Syntax:	[*label*] NOP
Operands:	None
Operation:	No operation
Status Affected:	None
Description:	No operation.

- Команда **nop** (No Operation) не змінює стан мікроконтролера
- Команда **nop** виконується автоматично у випадках порушення конвеєру замість команди, яку мікроконтролер підготував для виконання
- Команду **nop** використовують за необхідності формування потрібних затримок у часі

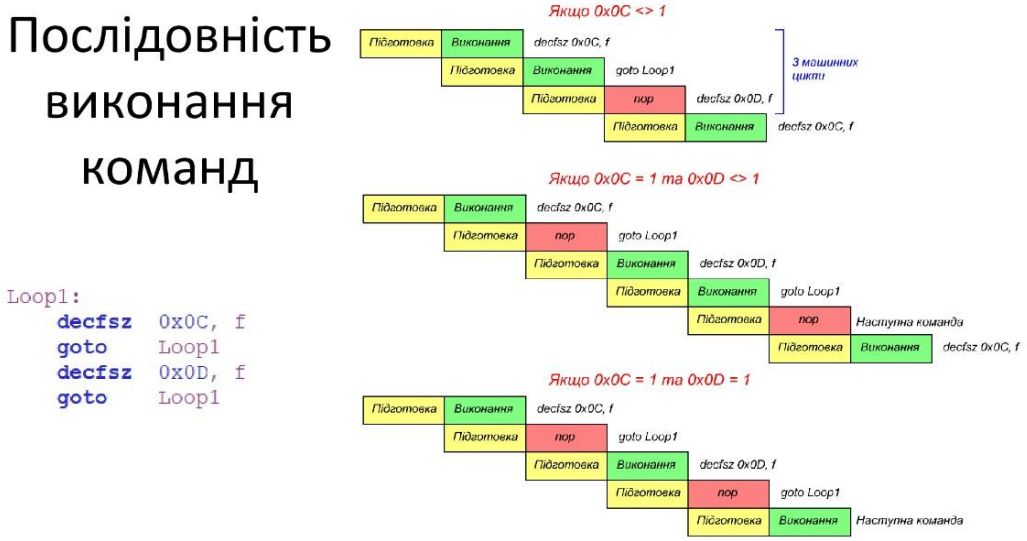
Розглянемо роботу коду затримки, в якому змінюються комірки пам’яті з адресами 0x0C і 0x 0D. Проаналізуємо послідовність виконання команд усередині мікроконтролера.

Коли в комірці пам’яті з адресою 0x 0C зберігається значення, що не дорівнює одиниці, спочатку виконується команда декременту. Паралельно з її виконанням відбувається підготовка до виконання команди переходу goto Loop1. Одночасно мікроконтролер готується до виконання команди декременту іншої комірки пам’яті. Оскільки після декременту результат не дорівнює нулю, команда переходу повинна виконуватися, але через порушення роботи конвеєра підготовлена наступна команда анулюється і замість неї виконується NOP. Після цього відбувається перехід назад до декременту комірки 0C. Таким чином, внутрішній цикл у цьому випадку займає три машинні цикли.

Коли ж у комірці пам’яті з адресою 0C зберігається одиниця, після декременту утворюється нуль. На апаратному рівні це означає, що наступну команду виконувати не можна, тому команда goto не виконується. Замість неї відповідний машинний цикл займає NOP. Паралельно відбувається підготовка до виконання декременту комірки з адресою 0D, який виконується аналогічно.

Якщо в обох комірках 0x0C і 0x0D міститься одиниця, то в обох випадках команди переходу анулюються і замінюються виконанням NOP. Після цього відбувається вихід із замкнутого циклу.

Таким чином, навіть простий на вигляд код затримки всередині мікроконтролера виконується з урахуванням роботи конвеєра, умовних переходів і анулювання команд, що безпосередньо впливає на фактичний час виконання.



Кількість машинних циклів, за які виконується конкретна команда, можна знайти в технічній документації, але ці значення не є абсолютними. Існують команди, для яких фактичний час виконання залежить від контексту.

Наприклад, у документації для команди ADDWF вказано виконання за один машинний цикл. Однак якщо як адресу призначення використати програмний лічильник, тобто фактично реалізувати перехід, еквівалентний goto, робота конвеєра порушується. У цьому випадку команда виконується вже за два машинні цикли.

Уся ця логіка реалізується на апаратному рівні мікроконтролера. У технічній документації для команд існує стовпчик Cycles, який показує кількість машинних циклів для стандартного випадку виконання інструкції.

Також видно, що, наприклад, команда декременту може займати один машинний цикл з переходом до наступної команди або два машинні цикли, де другий цикл фактично відповідає виконанню NOP у разі спрацювання умови. Це необхідно розуміти при аналізі часу виконання програм.

Тривалість виконання команди

TABLE 7-2: PIC16CXXX INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	0	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	0	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	0	0001	1fff	ffff	Z	2
CLRW	-	Clear W	1	0	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	0	1001	dfff	ffff	Z	1,2
DECF	f, d	Decrement f	1	0	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1 (2)	0	[illegible]				
INCF	f, d	Increment f	1	0	[illegible]				
INCFSZ	f, d	Increment f, Skip if 0	1 (2)	0	[illegible]				
IORWF	f, d	Inclusive OR W with f	1	0	[illegible]				
MOVF	f, d	Move f	1	0	[illegible]				
MOVWF	f	Move W to f	1	0	[illegible]				
NOP	-	No Operation	1	0	[illegible]				
RLF	f, d	Rotate Left f through Carry	1	0	[illegible]				
RRF	f, d	Rotate Right f through Carry	1	0	1100	dfff	ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	0	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	0	1110	dfff	ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	0	0110	dfff	ffff	Z	1,2

Тривалість виконання кожної команди (в машинних циклах) наведено в технічній документації на мікроконтролер

Тепер можна перейти до підрахунку тривалості виконання коду. Поки що розрахунки виконуються умовно, у машинних циклах, оскільки іншої інформації на даному етапі немає.

Розглядається фрагмент коду, в якому використовується одна комірка пам'яті. Команди movlw і movwf виконуються по одному машинному циклу кожна. Далі 255 разів виконується зв'язка з команди декременту та команди goto. У цій парі декремент займає один машинний цикл, а goto — два машинні цикли. Після завершення циклу виконується ще один декремент, після якого замість переходу виконується команда NOP.

Після підрахунку загальної кількості машинних циклів отримується значення 769. Саме стільки машинних циклів займає виконання цього фрагмента коду.

Тривалість виконання коду

```
    movlw   d'0'     ; 1 цикл
    movwf   0x0C     ; 1 цикл

Loop1:
    decfsz  0x0C, f ; 1 (2) цикл
    goto    Loop1    ; 2 цикли
```

movlw + movwf + 255 * (decfsz + goto) + (decfsz + nop)

1 + 1 + 255 * (1 + 2) + (1 + 1) = 769 циклів

Розглянемо інший приклад. У цьому випадку в комірку пам'яті 0C записується нуль, а перед цим у відповідну комірку записується одиниця. За таких умов код виконується лише за чотири машинні цикли.

У попередньому прикладі виконання займало 769 машинних циклів, а тут – чотири. Час виконання безпосередньо залежить від значень, записаних у комірки пам'яті. Таким чином стає зрозуміло, що тривалість затримки можна регулювати, змінюючи початкові дані.

Наприклад, якщо записати в комірку пам'яті значення 15, то весь цей код буде виконуватися за 46 машинних циклів. Саме так формується часовий інтервал шляхом підбору значень і підрахунку машинних циклів.

Тривалість виконання коду

```
    movlw   d'15'    ; 1 цикл
    movwf   0x0C     ; 1 цикл

Loop1:
    decfsz  0x0C, f ; 1 (2) цикл
    goto    Loop1    ; 2 цикли
```

movlw + movwf + 14 * (decfsz + goto) + (decfsz + nop)

1 + 1 + 14 * (1 + 2) + 2 = 46 циклів

Розглянемо більш складну конструкцію, у якій використовується знайомий цикл затримки, що застосовувався ще з перших практичних робіт. Тепер цей цикл можна точно проаналізувати і обчислити тривалість його виконання.

За точними підрахунками такий код виконується за 196 352 машинні цикли за умови, що на момент початку роботи в комірках пам'яті 0C і 0D містяться нулі. Якщо в цих комірках знаходяться інші значення, час виконання буде дещо меншим.

Програма побудована таким чином, що після завершення цього циклу в комірках 0C і 0D автоматично залишаються нульові значення. Винятком є перше виконання після перезавантаження, коли затримка може відрізнятися. Це пояснюється тим, що оперативна пам'ять при перезавантаженні не очищується і містить випадкові значення або значення, що залишилися після попереднього виконання програми. Це справедливо як для програмного перезавантаження, так і для перезавантаження кнопкою або подачі живлення. Виняток становлять лише деякі регістри спеціального призначення, які примусово встановлюються у визначені значення.

Таким чином, відповіддю на питання про тривалість виконання цього циклу є 196 352 машинні цикли.

Давайте розглянемо приклад збільшення тривалості виконання коду за допомогою команди NOP. Для цього у внутрішній цикл додаються кілька команд NOP, які не змінюють стан мікроконтролера, але збільшують час виконання. У результаті тривалість затримки зростає до 457 472 машинних циклів. Таким чином можна збільшувати час затримки, якщо стандартного циклу недостатньо.

Додавання команд NOP має сенс саме у внутрішньому циклі. У деяких випадках необхідно формувати дуже малі часові інтервали, що вимірюються мікросекундами, і тоді єдиним способом отримати потрібну затримку є використання порожніх команд.

На цьому етапі стає зрозумілим принцип підрахунку. Тепер можна точно обчислювати тривалість виконання будь-якого фрагмента коду в машинних циклах, не лише затримок. Залишається наступне питання: яка реальна тривалість одного машинного циклу в часі?

Тривалість виконання коду

```
Loop1:
    nop
    nop
    nop
    nop
    decfsz  0x0C, f
    goto    Loop1
    decfsz  0x0D, f
    goto    Loop1
```

255 * (255 * (4 * nop + decfsz 0x0C + goto) +
+ (4 * nop + decfsz 0x0C + nop) +
+ decfsz 0x0D + goto) + (decfsz 0x0D + nop)

255 * (255 * (4 * 1 + 1 + 2) +
+ (4 * 1 + 1 + 1) +
+ 1 + 2) + (1 + 1) = 457 472 циклів

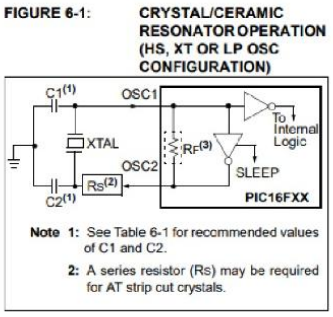
Тривалість одного машинного циклу залежить від схеми тактового генератора. Це питання в сучасних мікроконтролерах є доволі складним, особливо з огляду на вимоги до пристроїв, що живляться від батарей, де потрібна гнучка система тактування. Розглядуваний мікроконтролер PIC16F84 до таких складних систем не належить, тому аналізується лише те, що реалізовано в ньому.

Мікроконтролер може працювати в кількох режимах. Перший варіант — робота без власного генератора, коли мікроконтролер тактується від зовнішнього джерела. Такий режим застосовується в системах з кількома мікроконтролерами, які мають працювати синхронно від одного тактового сигналу. У цьому випадку зовнішній тактовий сигнал подається на вхід OSC1, і мікроконтролер працює від нього без додаткових налаштувань. Власний генератор при цьому не формується, а внутрішня схема лише підсилює сигнал, який за потреби може бути переданий далі.

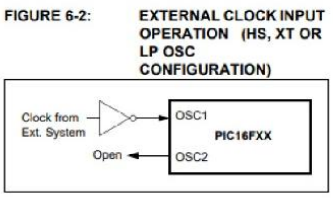
Другий варіант полягає у формуванні тактового сигналу за допомогою двох зовнішніх компонентів – резистора та конденсатора. Вони підключаються до виводу OSC1, а на виході OSC2 формується тактовий сигнал. Перевагою такого способу є низька вартість компонентів, однак на цьому переваги фактично закінчуються. Тактова частота є нестабільною і залежить від багатьох факторів, тому такий спосіб застосовується лише там, де не потрібна висока точність.

Найчастіше стабілізація тактової частоти здійснюється за допомогою кварцового або керамічного резонатора, які забезпечують значно кращу стабільність роботи.

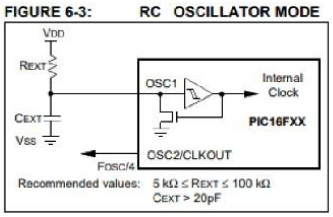
Варіанти тактового генератора



Стабілізація частоти за допомогою кварцового або керамічного резонатора



Використання зовнішнього джерела тактового сигналу (коли треба точно синхронізувати кілька мікроконтролерів)



Завдання частоти за допомогою зовнішнього RC-ланцюжка

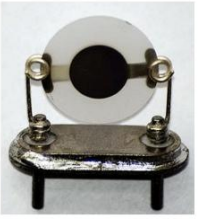
Тому розглянемо кварцовий резонатор. Існує мінерал кварц, який має властивість змінювати свої геометричні розміри залежно від прикладеної різниці потенціалів. Такі кристали належать до класу п’єзоелектриків. При механічній деформації на їхніх виводах з’являється різниця потенціалів, а при подачі напруги кристал починає деформуватися. Цю властивість можна використовувати для датчиків, але найчастіше кварц застосовується для стабілізації коливань.

Кварцовий резонатор є окремим електронним компонентом, який може мати різні конструктивні виконання. Усередині корпусу розміщується кристалотримач, у якому встановлений оброблений кварцовий кристал із заданими характеристиками.

Основна перевага кварцових резонаторів полягає у високій точності підтримання частоти. Відхилення частоти дуже малі, на рівні сотих часток відсотка. Для порівняння, в електроніці відхилення у 10 % вважається допустимим, тоді як для кварцу характерна значно вища точність. Крім того, частота коливань кварцових резонаторів слабо залежить від температури, напруги живлення та інших зовнішніх факторів, попри те, що самі коливання мають механічну природу.

Таким чином, найкращим способом стабілізації частоти є використання кварцевого резонатора. Його недоліком є вища вартість порівняно з резисторами або конденсаторами, однак у схемі зазвичай використовується лише один такий елемент.

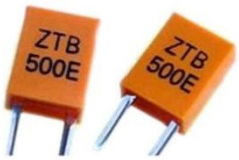
Кварцовий резонатор



- Стабілізація частоти відбувається за допомогою пластини кварцу, в якій виникають резонансні механічні коливання на певних частотах
- Найвища стабільність частоти
- Найдорожчий із стабілізаторів частоти
- Потрібні додаткові електронні компоненти (конденсатори, інколи – резистори)
- Використання – застосунки, що потребують високоточної роботи у часі (годинники, системи зв’язку, тощо)

Керамічні резонатори є дешевшою альтернативою кварцовим. Вони працюють за тим самим принципом, але замість природного кварцу використовується кераміка. Стабільність керамічних резонаторів нижча, проте їх застосовують у тих випадках, де висока точність частоти не є критичною, завдяки меншій вартості.

Керамічний резонатор



- Стабілізація частоти відбувається за допомогою пластини з кераміки, в якій виникають резонансні коливання на певних частотах
- Середня стабільність частоти
- Середня вартість
- Інколи потрібні додаткові компоненти
- Використання – застосунки, що не потребують високоточної роботи у часі (наприклад, пульти дистанційного керування)

RC-генератор є найдешевшим варіантом тактування. Його частота залежить від параметрів електронних компонентів, самого мікроконтролера та температури. Навіть при однакових резисторах і конденсаторах заміна мікроконтролера може призвести до зміни частоти через технологічний розкид внутрішнього генератора. Температурні зміни, особливо для керамічних конденсаторів, також суттєво впливають на ємність і, відповідно, на частоту.

Основною перевагою RC-генератора є його низька вартість. У сучасних мікроконтролерах RC-генератори зазвичай вбудовані та не потребують зовнішніх компонентів. Невеликі ємності можуть бути реалізовані безпосередньо на кристалі, тоді як індуктивності та конденсатори загалом складно формувати всередині мікросхем, на відміну від резисторів. Саме тому більшість сучасних мікроконтролерів мають вбудований тактовий генератор, і після перезавантаження ядро за замовчуванням починає працювати саме від нього.

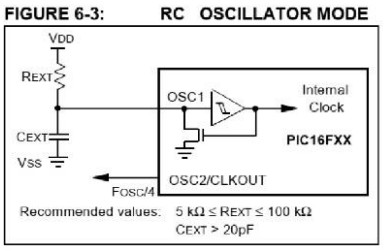
Таке рішення значно спрощує запуск системи. На практиці кварцовий резонатор не завжди вдається запустити з першого разу. Кварци різних виробників можуть по-різному працювати з тим самим мікроконтролером, а класична кварцова схема додатково залежить від параметрів зовнішніх конденсаторів. Якщо тактовий генератор не запускається, мікроконтролер не подає жодних ознак життя, і програміст не може зрозуміти, у чому причина проблеми, оскільки програма не стартує без тактування.

Навіть у серійному виробництві з кварцовими резонаторами виникають труднощі на етапі першого запуску. Саме тому виробники мікроконтролерів пішли шляхом використання внутрішнього RC-генератора як стартового. Він працює в будь-якому випадку, програма гарантовано запускається, після чого програмно можна спробувати ввімкнути зовнішній тактовий генератор. Через регістри спеціального призначення можна перевірити, чи запрацював кварц, і в разі відмови передати діагностичну інформацію, наприклад через порти введення-виведення.

Таким чином, попри низьку стабільність і точність, RC-генератор є важливим інструментом, який суттєво полегшує розробку та налагодження програмного забезпечення. Залишається питання: яким чином налаштувати мікроконтролер так, щоб його тактовий генератор працював у потрібному режимі?

RC-генератор

FIGURE 6-3: RC OSCILLATOR MODE



Більшість сучасних мікроконтролерів має вбудований RC-генератор для роботи якого не потрібні зовнішні компоненти

- Частота роботи задається двома зовнішніми компонентами (резистором та конденсатором)
- Нестабільна частота, що залежить від напруги живлення, температури, старіння елементів тощо
- У різних пристроїв частота може значно відрізнятися через технологічний розкид параметрів резистора, конденсатора та мікроконтролера
- Найнижча вартість
- Використання – застосунки, де частота виконання програмного коду може змінюватися в широких межах (побутова автоматика, ялинкові гірлянди тощо)

Тепер підходимо до останнього елемента програми Hello World, який раніше не розглядався – конфігурації. Вона може називатися configuration word, configuration bytes або fuses. Це спеціальна частина пам’яті програм, яка недоступна для зміни під час виконання програми.

Слово конфігурації розташоване за адресою поза межами адресного простору мікроконтролера. Його можна змінити лише на етапі програмування мікроконтролера. Після запуску програми змінити ці параметри вже неможливо.

У мікроконтролері PIC16F84 використовується одне слово конфігурації. У більш сучасних мікроконтролерах таких слів може бути декілька. У слові конфігурації задаються

основні параметри роботи мікроконтролера: режим роботи тактового генератора, стан сторожового таймера, таймер затримки подачі живлення та захист програми від зчитування.

Усі ці параметри програмуються на етапі прошивки, але безпосередньо з програмного коду вони недоступні.

Слово конфігурації

REGISTER 6-1: PIC16F84A CONFIGURATION WORD

R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u
CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	$\overline{\text{PWRTE}}$	WDTE	F0SC1	F0SC0
bit13													bit0

bit 13-4 **CP:** Code Protection bit
1 = Code protection disabled
0 = All program memory is code protected

bit 3 **$\overline{\text{PWRTE}}$**: Power-up Timer Enable bit
1 = Power-up Timer is disabled
0 = Power-up Timer is enabled

bit 2 **WDTE**: Watchdog Timer Enable bit
1 = WDT enabled
0 = WDT disabled

bit 1-0 **FOSC1:FOSC0**: Oscillator Selection bits
11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

Слово конфігурації (Configuration Word, Configuration Bytes (Bits) або Fuses) містить налаштування мікроконтролера, які не можна змінити програмним шляхом

Слово конфігурації є частиною пам’яті програм, але адреса цієї комірки (0x2007) знаходиться поза межами адресного простору, до якого можна звернутися за допомогою команд мови асемблеру

Коротко розглянемо функції, які задаються в слові конфігурації. Захист програмного коду призначений для запобігання несанкціонованому копіюванню прошивки. Після запису програми в мікроконтролер і встановлення його в готовий пристрій існує ризик, що хтось зчитає пам’ять програм і отримає готовий файл прошивки, який можна безпосередньо записати в інший мікроконтролер без будь-якого аналізу коду.

Для цього передбачені біти захисту. При їх встановленні зчитування пам’яті програм стає неможливим. У деяких мікроконтролерах існують окремі біти захисту для пам’яті програм і пам’яті даних, але в цьому мікроконтролері використовується лише один біт захисту, хоча загалом таких бітів може бути декілька.

Якщо біти захисту встановлені, мікроконтролер продовжує нормально працювати, але зчитати з нього програму неможливо. Перезапис можливий лише через спеціальну процедуру повного очищення. Після очищення мікроконтролер повертається до заводського стану, вміст пам’яті програм стає недоступним для читання і виглядає як набір одиниць. Після цього мікроконтролер можна знову прошити новою програмою і повторно встановити біт захисту.

Захист програмного коду

bit 13-4 **CP:** Code Protection bit
1 = Code protection disabled
0 = All program memory is code protected

6.12 Program Verification/Code Protection

If the code protection bit(s) have not been programmed, the on-chip program memory can be read out for verification purposes.

- Захист програмного коду призначений для запобігання крадіжок програмного забезпечення (прошивки) з серійних виробів
- Якщо біти CP встановлені в значення 0, пам’ять програм не можна прочитати
- Після встановлення бітів CP в значення 0 змінити пам’ять програм в мікроконтролері можна лише після повного стирання інформації (при цьому усі біти у пам’яті програм, зокрема і слова конфігурації, будуть встановлені в значення 1)

Розглянемо затримку запуску. Мікроконтролер працює в режимі реального часу і керує зовнішніми електронними схемами. Він починає виконання програми дуже швидко, і може виникнути ситуація, коли мікроконтролер уже працює, а зовнішня схема ще не готова.

Для цього мікроконтролер примусово утримується в стані перезавантаження протягом певного часу. У кожного мікроконтролера є вивід скидання, який у даному випадку називається MCLR (Master Clear). Сигнал скидання має активний низький рівень.

Щоб мікроконтролер не починав роботу одразу після подачі живлення, до цього виводу підключається RC-ланцюжок з резистора і конденсатора. У момент подачі живлення конденсатор розряджений, на вході скидання присутній логічний нуль, і мікроконтролер утримується в стані reset. У міру заряджання конденсатора напруга зростає, і через заданий час досягає рівня логічної одиниці, після чого мікроконтролер починає роботу. Тривалість цієї затримки визначається параметрами RC-ланцюжка.

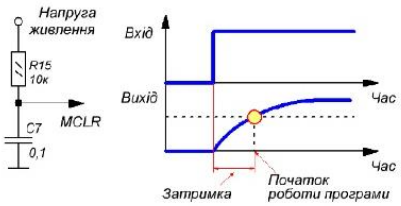
У мікроконтролері PIC16F84 також є внутрішній таймер, який дозволяє сформувати затримку запуску приблизно 72 мілісекунди без використання будь-яких зовнішніх компонентів.

Таймер затримки запуску мікроконтролера

bit 3 — $\overline{\text{PWRTE}}$: Power-up Timer Enable bit
1 = Power-up Timer is disabled
0 = Power-up Timer is enabled

6.5 Power-up Timer (PWRT)

The Power-up Timer (PWRT) provides a fixed 72 ms nominal time-out (TPWRT) from POR (Figures 6-6 through 6-9). The Power-up Timer operates on an internal RC oscillator. The chip is kept in RESET as long as the PWRT is active. The PWRT delay allows the VDD to rise to an acceptable level (possible exception shown in Figure 6-9).



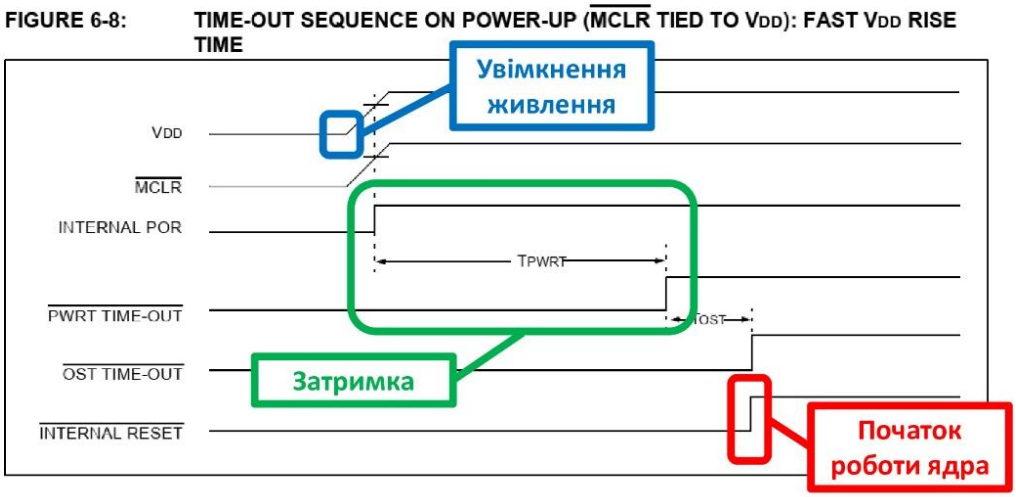
- Таймер затримки запуску мікроконтролера (Power-up Timer) утримує мікроконтролер у стані перезавантаження (RESET) впродовж приблизно 72 мс після подачі напруги живлення достатньої якості
- Таймер затримки запуску дозволяє сформувати затримку без використання зовнішніх компонентів (здешевити пристрій)
- Затримка запуску мікроконтролера потрібна для того, щоб програма (мікроконтролер) почала працювати після завершення перехідних процесів у зовнішній електричній схемі

Розглянемо, як це працює на практиці. У цьому випадку вивід MCLR підключений безпосередньо до виводу живлення VDD. Після подачі живлення напруга зростає не миттєво, а протягом певного часу. Коли напруга досягає рівня, за якого внутрішня логіка може працювати, першим вмикається внутрішній таймер затримки.

Цей таймер приблизно 72 мс утримує мікроконтролер у стані reset, навіть якщо його апаратна частина вже частково працездатна. Після цього додається ще одна апаратна затримка, необхідна для коректного запуску тактового генератора, оскільки він також не виходить одразу на робочий режим. Лише після завершення цих етапів починає працювати ядро мікроконтролера, і стартує виконання першої команди програми.

У результаті між моментом подачі живлення і початком виконання програми існує достатньо великий часовий інтервал, необхідний для того, щоб зовнішні електронні схеми встигли перейти в робочий стан. Це важливо, оскільки на початку програми часто аналізується стан зовнішніх модулів. Якщо живлення подано, а зовнішній модуль ще не готовий і сигналізує про помилку, система може прийняти неправильне рішення. Саме в таких випадках таймер затримки запуску є корисним і дозволяє уникнути помилкової роботи системи.

Таймер затримки запуску мікроконтролера

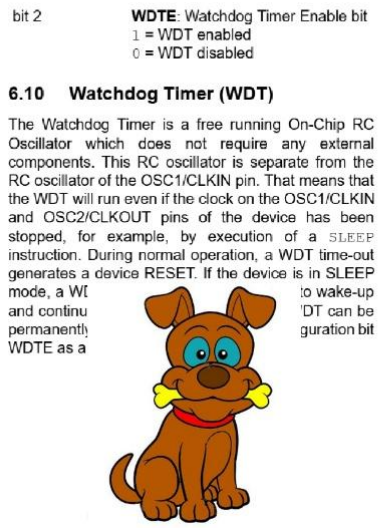


Сторожовий таймер (дослівно: собака, що споглядає) – це автономний апаратний модуль мікроконтролера, який має власний внутрішній генератор і працює незалежно від програмного коду та основного тактового генератора. Його завдання – перезавантажувати мікроконтролер у разі, якщо програмне забезпечення перестає працювати коректно.

Принцип роботи пояснюється аналогією з собакою. Якщо цю «собаку» вчасно не годувати, вона перезавантажує все програмне забезпечення. Годуванням є спеціальна команда в програмі, яка регулярно повідомляє сторожовому таймеру, що програма працює нормально. Якщо команда не виконується вчасно, таймер вважає, що програма зависла, і виконує reset.

Сторожовий таймер можна не використовувати, але якщо його активовано, вимкнути його програмно в цьому мікроконтролері неможливо. Тому в програмі обов'язково потрібно передбачити регулярне «годування» цієї собаки, інакше мікроконтролер буде постійно перезавантажуватися.

Сторожовий таймер


bit 2 **WDTE**: Watchdog Timer Enable bit
1 = WDT enabled
0 = WDT disabled

6.10 Watchdog Timer (WDT)

The Watchdog Timer is a free running On-Chip RC Oscillator which does not require any external components. This RC oscillator is separate from the RC oscillator of the OSC1/CLKIN pin. That means that the WDT will run even if the clock on the OSC1/CLKIN and OSC2/CLKOUT pins of the device has been stopped, for example, by execution of a SLEEP instruction. During normal operation, a WDT time-out generates a device RESET. If the device is in SLEEP mode, a WI to wake-up and continu DT can be permanently guration bit WDTE as a

- Сторожовий таймер (Watchdog Timer, WDT) є повністю автономним апаратним модулем, який на апаратному рівні слідкує за працездатністю програмного забезпечення
- Якщо програмне забезпечення зависло або працює неправильно, то сторожовий таймер автоматично перезавантажить мікроконтролер
- Використання сторожового таймеру дозволяє підвищити життєздатність застосунку
- При створенні програмного забезпечення з використанням сторожового таймеру програмісту потрібно періодично його перезавантажувати, щоб уникнути перезавантаження мікроконтролера

Для цього передбачена спеціальна команда CLRWDT, тобто очищення лічильника сторожового таймера. Сам сторожовий таймер має діапазон спрацювання приблизно від 18 мікросекунд до 2,3 секунди. Якщо у програмі використовується сторожовий таймер, необхідно передбачити регулярний виклик команди CLRWDT, тобто періодично «годувати собаку».

Поки команда CLRWDT виконується вчасно, сторожовий таймер не спрацьовує. Як тільки програма зависає і ця команда перестає виконуватися, таймер автоматично перезавантажує програмне забезпечення.

Зависання програми може мати різні причини: логічні помилки в коді, зациклення, переповнення стеку або спотворення вмісту оперативної пам'яті, наприклад, через зовнішні завади, такі як, розряд блискавки. У таких випадках програма починає працювати з некоректними даними і поводиться не так, як було заплановано.

Сторожовий таймер у таких ситуаціях автоматично перезавантажує мікроконтролер і відновлює нормальну роботу системи, тому ця функція є дуже корисною.

Робота зі сторожовим таймером

CLRWDT	Clear Watchdog Timer
Syntax:	[*label*] CLRWDT
Operands:	None
Operation:	00h → WDT 0 → WDT prescaler, 1 → $\overline{TO}$ 1 → $\overline{PD}$
Status Affected:	$\overline{TO}$, $\overline{PD}$
Description:	CLRWDT instruction resets the Watchdog Timer. It also resets the prescaler of the WDT. Status bits $\overline{TO}$ and $\overline{PD}$ are set.

- Після активації та налаштування сторожовий таймер буде перезавантажувати мікроконтролер через певні проміжки часу, що налаштовуються (від 0,018 с до 2,3 с)
- Для запобігання перезавантаженню використовується спеціальна асемблерна команда **clrwdt**, яка очищує внутрішні лічильники сторожового таймеру
- При створенні програмного забезпечення з активним сторожовим таймером програміст повинен слідкувати за часом виконання програмного коду та додавати у код команду **clrwdt**

Останні два біти слова конфігурації задають режим роботи тактового генератора. Усього передбачено чотири режими. Перший режим – RC. Інші режими залежать від робочої частоти генератора, оскільки неможливо створити універсальний генератор, який стабільно працював би на всіх частотах.

Для низьких частот використовується режим LP, для середніх – режим XT, для високих – режим HS. Вибір режиму визначається робочою частотою, при цьому чітких жорстких меж не існує. Однак на практиці, наприклад, у режимі LP генератор не зможе запуститися на високих частотах, таких як 20 МГц. На апаратному рівні це пов'язано з параметрами збудження резонатора: струм буде недостатнім, у результаті генератор або не запуститься взагалі, або працюватиме нестабільно.

Тактовий генератор

bit 1-0 **FOSC1:FOSC0**: Oscillator Selection bits
11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

6.2.1 OSCILLATOR TYPES

The PIC16F84A can be operated in four different oscillator modes. The user can program two configuration bits (FOSC1 and FOSC0) to select one of these four modes:

- LP Low Power Crystal
- XT Crystal/Resonator
- HS High Speed Crystal/Resonator
- RC Resistor/Capacitor

- **RC** – частота задається зовнішніми резистором та конденсатором
- **HS** – частота задається високочастотним кварцовим або керамічним резонатором (4…20 МГц)
- **XT** – частота задається кварцовим або керамічним резонатором, що працює на середніх частотах (0,1…4 МГц)
- **LP** – частота задається низькочастотним кварцовим резонатором (0…0,2 МГц)

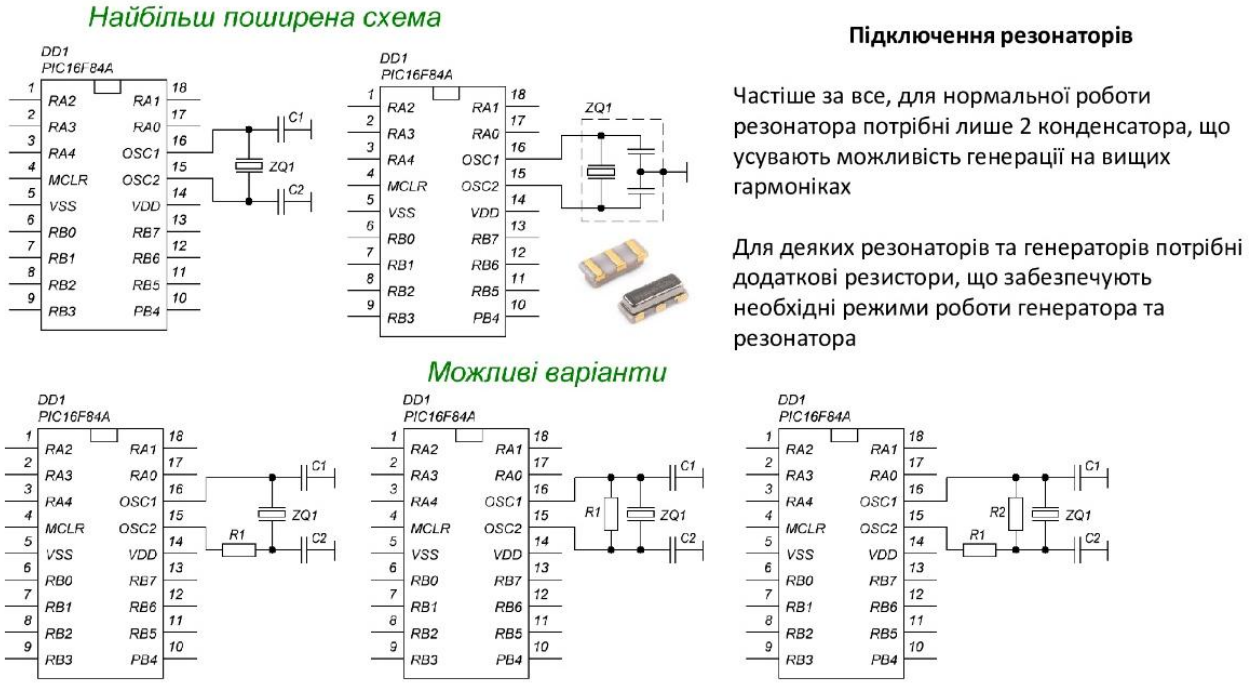
Неправильне встановлення бітів, що відповідають за налаштування тактового генератора, призведуть до того, що він не буде працювати або буде працювати нестабільно

Розглянемо самі резонатори. Найпоширеніша схема їх використання є класичною. Застосовується резонатор, кварцовий або керамічний, що в даному випадку не має принципового значення, і два конденсатори, які підключаються від його виводів до спільного проводу. Ця схема настільки поширена, що існують готові компоненти, у яких кварц і необхідні конденсатори вже розміщені в одному корпусі.

Така популярність пояснюється простотою та зручністю: кварц усе одно не використовується без конденсаторів, а значення їх ємностей зазвичай відомі і залежать від робочої частоти. Це зменшує кількість зовнішніх компонентів і спрощує виробництво. Водночас можливі варіанти, коли потрібні інші значення ємностей, наприклад при нестандартних частотах, дуже низьких або дуже високих, або коли кварц має специфічну конструкцію.

У деяких випадках, окрім конденсаторів, можуть використовуватися додаткові резистори, які вмикаються послідовно або паралельно з кварцом. Такі рішення залежать не стільки від документації на мікроконтролер, скільки від характеристик самого резонатора, тому необхідно звертатися до документації на кварц.

Кварц виготовляється з природних мінералів, і родовища в різних країнах мають відмінний хімічний склад. Через це кварци від різних виробників можуть мати дещо різні властивості, що інколи вимагає додаткових елементів у схемі. Тому розробники часто передбачають на платі можливість встановлення таких резисторів, навіть якщо в більшості випадків вони не потрібні.



Ємності конденсаторів для кварцевого резонатора зазвичай ніхто точно не розраховує. Хоча існують відповідні теоретичні дослідження, реальні розрахунки настільки складні з точки зору математики й фізики, що на практиці все зводиться до вибору ємностей у певному рекомендованому діапазоні залежно від частоти кварцу. Найчастіше розробник просто обирає середнє значення з цього діапазону.

Якщо під час налагодження або серійного виробництва виникають проблеми, тоді ємності коригуються: їх збільшують або зменшують, змінюють тип діелектрика, оскільки він також впливає на роботу генератора. Кварцові резонатори є досить капризними компонентами, і перший запуск пристрою часто є показовим. Якщо після подачі живлення система одразу починає працювати, це означає, що схема зібрана коректно. Якщо ж нічого не працює, доводиться перевіряти живлення, сигнал скидання та роботу кварцу.

Для режиму RC взагалі не існує точних формул розрахунку частоти, оскільки вона залежить від багатьох взаємопов'язаних параметрів. У документації зазвичай наводяться лише орієнтовні залежності. Тому ключовим залишається уважне читання технічної документації, оскільки навіть у стандартних схемах можуть існувати нюанси, які впливають на роботу системи.

Параметри електронних компонентів

TABLE 6-1: CAPACITOR SELECTION FOR CERAMIC RESONATORS

Ranges Tested:			
Mode	**Freq**	**OSC1/C1**	**OSC2/C2**
XT	455 kHz	47 - 100 pF	47 - 100 pF
	2.0 MHz	15 - 33 pF	15 - 33 pF
	4.0 MHz	15 - 33 pF	15 - 33 pF
HS	8.0 MHz	15 - 33 pF	15 - 33 pF
	10.0 MHz	15 - 33 pF	15 - 33 pF

Note: Recommended values of C1 and C2 are identical to the ranges tested in this table. Higher capacitance increases the stability of the oscillator, but also increases the start-up time. These values are for design guidance only. Since each resonator has its own characteristics, the user should consult the resonator manufacturer for the appropriate values of external components.

Note: When using resonators with frequencies above 3.5 MHz, the use of HS mode rather than XT mode, is recommended. HS mode may be used at any VDD for which the controller is rated.

TABLE 6-2: CAPACITOR SELECTION FOR CRYSTAL OSCILLATOR

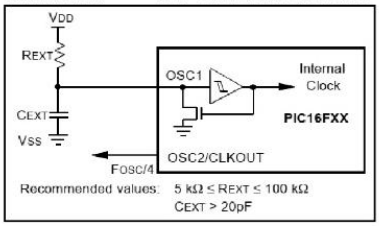
Mode	Freq	OSC1/C1	OSC2/C2
LP	32 kHz	68 - 100 pF	68 - 100 pF
	200 kHz	15 - 33 pF	15 - 33 pF
XT	100 kHz	100 - 150 pF	100 - 150 pF
	2 MHz	15 - 33 pF	15 - 33 pF
	4 MHz	15 - 33 pF	15 - 33 pF
HS	4 MHz	15 - 33 pF	15 - 33 pF
	20 MHz	15 - 33 pF	15 - 33 pF

Note: Higher capacitance increases the stability of the oscillator, but also increases the start-up time. These values are for design guidance only. Rs may be required in HS mode, as well as XT mode, to avoid over-driving crystals with low drive level specification. Since each crystal has its own characteristics, the user should consult the crystal manufacturer for appropriate values of external components.
For VDD > 4.5V, C1 = C2 ≈ 30 pF is recommended.

6.2.3 RC OSCILLATOR

For timing insensitive applications, the RC device option offers additional cost savings. The RC oscillator frequency is a function of the supply voltage, the resistor (REXT) values, capacitor (CEXT) values, and the operating temperature. In addition to this, the oscillator frequency will vary from unit to unit due to normal process parameter variation. Furthermore, the difference in lead frame capacitance between package types also affects the oscillation frequency, especially for low CEXT values. The user needs to take into account variation, due to tolerance of the external R and C components. Figure 6-3 shows how an R/C combination is connected to the PIC16F84A.

FIGURE 6-3: RC OSCILLATOR MODE



Отже, слово конфігурації. За замовчуванням флеш-пам'ять після стирання заповнюється одиницями, тобто в усі біти записуються логічні одиниці. Відповідно слово конфігурації також складається з одиниць. Його довжина є нестандартною і становить 14 біт, а не 8 чи 16.

У такому стані захист коду вимкнений, таймер затримки запуску вимкнений, сторожовий таймер увімкнений і працює, а режим роботи тактового генератора встановлений за замовчуванням. Якщо необхідно, щоб програма працювала з кварцовим резонатором, слово конфігурації потрібно змінювати на етапі програмування.

Початкове значення слова конфігурації

R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u
CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	$\overline{\text{PWRTE}}$	WDTE	F0SC1	F0SC0
bit13													bit0

Після очищення Flash-пам'яті усі її комірки отримують електричний заряд, наявність якого інтерпретується як логічна одиниця

Початкове значення слова конфігурації – b'11111111111111' (0x3FFF)

- Захист коду (CP) – вимкнено
- Таймер затримки запуску (PWRT) – вимкнено
- Сторожовий таймер (WDT) – увімкнено
- Режим роботи тактового генератора – RC-генератор

Якщо залишити слово конфігурації без змін, то мікроконтролер, до якого підключений кварцовий або керамічний резонатор, не буде подавати жодних «ознак життя»

Налаштування слова конфігурації виконується безпосередньо в програмному коді. Це, як правило, перший рядок програми, у якому використовується директива CONFIG. Вона визначає, за якою адресою і з яким значенням буде записане слово конфігурації.

Усі необхідні константи для формування цього слова описані у відповідному файлі мікроконтролера, зокрема для P16F74. За допомогою побітових операцій з цих констант формується одне числове значення, яке і задає слово конфігурації. Саме через директиву CONFIG у програмі визначаються всі параметри конфігурації мікроконтролера.

З цими параметрами можна експериментувати, щоб побачити, як змінюється робота програми. Наприклад, якщо увімкнути сторожовий таймер і не передбачити його обслуговування, програма почне постійно перезавантажуватися. У такій ситуації, не знаючи про сторожовий таймер, можна довго шукати помилку в коді, хоча насправді працює сторожова собака, яку необхідно регулярно годувати.

Як налаштувати слово конфігурації

```
;==============================================================================
;
;       Configuration Bits
;
;   NAME            Address
;   CONFIG            2007h
;
;==============================================================================

; The following is an assignment of address values for all of the
; configuration registers for the purpose of table reads
_CONFIG         EQU  H'2007'

;----- CONFIG Options ----------------------------------------------
_FOSC_LP            EQU  H'3FFC'    ; LP oscillator
_LP_OSC             EQU  H'3FFC'    ; LP oscillator
_FOSC_XT            EQU  H'3FFD'    ; XT oscillator
_XT_OSC             EQU  H'3FFD'    ; XT oscillator
_FOSC_HS            EQU  H'3FFE'    ; HS oscillator
_HS_OSC             EQU  H'3FFE'    ; HS oscillator
_FOSC_EXTRC         EQU  H'3FFF'    ; RC oscillator
_RC_OSC             EQU  H'3FFF'    ; RC oscillator

_WDTE_OFF           EQU  H'3FFB'    ; WDT disabled
_WDT_OFF            EQU  H'3FFB'    ; WDT disabled
_WDTE_ON            EQU  H'3FFF'    ; WDT enabled
_WDT_ON             EQU  H'3FFF'    ; WDT enabled

_PWRTE_ON           EQU  H'3FF7'    ; Power-up Timer is enabled
_PWRTE_OFF          EQU  H'3FFF'    ; Power-up Timer is disabled

_CP_ON              EQU  H'000F'    ; All program memory is code protected
_CP_OFF             EQU  H'3FFF'    ; Code protection disabled
```

```
__CONFIG _XT_OSC & _WDT_OFF & _PWRTE_OFF & _CP_OFF
```

Слово конфігурації налаштовується за допомогою директиви **_CONFIG** шляхом об'єднання констант, визначених у файлі **p16F84A.inc** задопомогою логічної операції «**Побітове І**» (**&**)

- Захист коду (CP) – вимкнено (**_CP_OFF**)
- Таймер затримки запуску (PWRT) – вимкнено (**_PWRTE_OFF**)
- Сторожовий таймер (WDT) – вимкнено (**_WDT_OFF**)
- Режим роботи тактового генератора – XT (**_XT_OSC**)

Розглянемо, як це реалізується на практиці. Для формування слова конфігурації використовується логічна побітова операція **І**, яка еквівалентна логічному множенню. Якщо хоча б один із бітів дорівнює нулю, результат також буде нуль. Якщо всі біти дорівнюють одиниці, результатом буде одиниця.

Використовується набір констант. Константа CP_OFF складається з одиниць, тобто програмний код не захищається. Константа PWRTE_OFF також містить лише одиниці, і після побітової операції результат залишається без змін. У константі WDT_OFF відповідний біт дорівнює нулю, тому результат операції для цього біта також буде нуль. У константі XT_OSC нуль встановлений у першому біті. У підсумку слово конфігурації містить одиниці в усіх бітах, окрім першого та другого.

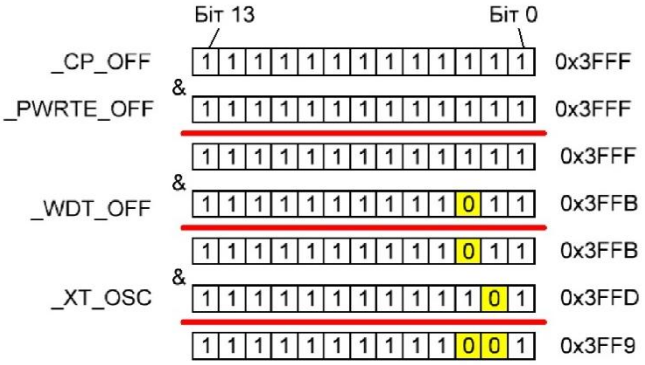
Сформоване таким чином слово конфігурації передається програматору. Спочатку завантажується пам'ять програм, потім перевіряється коректність запису, і лише після цього встановлюється слово конфігурації. Якщо в ньому активований захист коду, зчитати записану програму вже буде неможливо.

Формування слова конфігурації

__CONFIG _XT_OSC & _WDT_OFF & _PWRTE_OFF & _CP_OFF

Побітове І

X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1



Отже, програма виконується в машинних циклах. За тактової частоти 4 МГц за одну секунду відбувається 4 мільйони тактових коливань. Один машинний цикл складається з чотирьох тактових інтервалів, тому мікроконтролер виконує 1 мільйон команд за секунду. Відповідно, за частоти 4 МГц тривалість виконання однієї команди становить 1 мікросекунду.

ТАЙМЕРИ ТА ПЕРЕРИВАННЯ

Отже, продовжуємо вивчати предмет програмування мікроконтролерів та пристроїв інтернету речей. Тема сьогоднішньої лекції – таймери та переривання.

Мікроконтролер працює в реальному часі, і всі процеси, що відбуваються всередині нього, жорстко прив'язані до джерела тактового сигналу – тактового генератора. Саме цей генератор формує тактові імпульси, від яких залежить швидкість і перебіг усіх внутрішніх процесів.

На попередніх заняттях було створено кілька реальних програм і застосунків, у процесі яких відпрацьовувалися базові принципи програмування мікроконтролерів. Сьогодні знову розглядається перша навчальна програма «Вітаю, світе!». З одного боку наведений код на асемблері, з іншого – та сама програма, реалізована мовою **C**. Це демонструє, що принципи створення програмного забезпечення для мікроконтролерів не залежать від мови програмування. І на асемблері, і на **C** необхідно виконувати однакові дії: ініціалізувати мікроконтролер, створювати нескінченний основний цикл, використовувати підпрограми.

Програма «Вітаю, Світе!»

```
        org     0x000

        ; налаштування порту B
        bsf     STATUS, RP0
        movlw   b'11111110'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
        ; засвітити світлодіод
        bsf     PORTB, RB0
        call    Pause

        ; загасити світлодіод
        bcf     PORTB, RB0
        call    Pause

        goto    MainLoop

Pause:  ; затримка (пауза)
        incfsz  0x0C
        goto    Pause
        incfsz  0x0D
        goto    Pause
        return

        END
```

```
#include "stm8s.h"

void pause(void);

void main(void)
{
  //  ініціалізація
  GPIO_Init(GPIOB, GPIO_PIN_5, GPIO_MODE_OUT_OD_HIZ_SLOW);

  //  основний цикл
  while(1)
  {
    GPIO_WriteHigh(GPIOB, GPIO_PIN_5);
    pause();
    GPIO_WriteLow(GPIOB, GPIO_PIN_5);
    pause();
  }
}

// підпрограма паузи
void pause(void)
{
  for(uint16_t i = 0; i < 60000; i++) {}
}
```

На мовах високого рівня програмування ці дії виконувати простіше і швидше, з меншою ймовірністю помилок. Водночас асемблер дає повне розуміння того, який код буде виконуватися апаратно, тоді як результат компіляції мов високого рівня залежить від якості компілятора і того, як він транслює код у машинні інструкції.

Відомо, що мікроконтролер працює дуже швидко. Наприклад, PIC16F84 за тактової частоти 4 МГц виконує близько одного мільйона команд за секунду. Для людського сприйняття це надто швидко, тому, щоб спостерігати, як світлодіод засвічується і згасає, між цими подіями необхідно створити помітний часовий інтервал.

Цей інтервал формується підпрограмою затримки, яку умовно називають паузою. Фактично мікроконтролер навантажується марною з точки зору функціоналу роботою. У наведеному прикладі це багаторазове інкрементування двох комірок пам'яті з адресами 0x0C і 0x0D або циклічне збільшення змінної в заданому діапазоні значень. За рахунок цього процесорний час витрачається на виконання циклів, що і створює затримку в часі. Таким чином реалізується програмна пауза між змінами стану світлодіода.

Зрозуміло, що такий підхід є неефективним, оскільки процесорний час витрачається на роботу, яка не має корисного навантаження. Процесор міг би використовуватися для розв'язання інших задач, але замість цього виконує арифметико-логічні операції, результати яких ніде не застосовуються. Тому доцільно оцінити ефективність використання процесорного часу.

Швидкість виконання коду вимірюється в машинних циклах. Одна асемблерна команда може виконуватися за один або за два машинних цикли. Один машинний цикл відповідає чотирьом тактам тактового генератора. Ці принципи були розглянуті на попередніх лекціях.

У наведених розрахунках враховується лише основний цикл програми. Блок ініціалізації не розглядається, оскільки він виконується один раз і практично не підлягає оптимізації. Якщо мікроконтролер необхідно налаштувати, це потрібно зробити незалежно від бажання оптимізувати код.

Обчислення тривалості виконання основного циклу

```
        org     0x000

        ; налаштування порту B
        bsf     STATUS, RP0
        movlw   b'11111110'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
        ; засвітити світлодіод
        bsf     PORTB, RB0
        call    Pause

        ; загасити світлодіод
        bcf     PORTB, RB0
        call    Pause

        goto    MainLoop

Pause:  ; затримка (пауза)
        incfsz  0x0C
        goto    Pause
        incfsz  0x0D
        goto    Pause
        return

        END
```

Основний цикл

1 (**bsf**) + 2 (**call**) + 1 (**bcf**) + 2 (**call**) + 2 (**goto**) = 8 машинних циклів

Підпрограма паузи

255 * ([255 * (1 (**incfsz**) + 2 (**goto**)) + (1 (**incfsz**) + 1 (**nop**))] + 1 (**incfsz**) + 2 (**goto**)) +

+ (1 (**incfsz**) + 1 (**nop**)) + 2 (**return**) = 196 354 машинні цикли

Кількість «корисних» машинних циклів основного циклу

1 (**bsf**) + 1 (**bcf**) + 2 (**goto**) = 4 машинних циклів

Кількість машинних циклів, що формують затримку у часі

2 * (2 (**call**) + 196 354 (**підпрограма Pause**)) = 392 712 машинних цикли

Відносна кількість «корисних» команд в основному циклі
(коефіцієнт корисного використання процесорного часу)

4 / (392712 + 4) = 0,001%

У основному циклі корисних команд усього чотири. Це команди **bsf** і **bcf**. Одна команда **bsf** засвічує світлодіод, команда **bcf** його гасить. Також використовується команда **goto**, яка необхідна для організації нескінченного основного циклу і виконується за кілька машинних циклів. Усі інші команди в циклі відносяться до формування затримки.

Якщо підрахувати сумарну кількість команд, які виконує мікропроцесор для забезпечення затримки, то за наведеними розрахунками виходить 39712 машинних циклів. Загальна тривалість однієї ітерації основного циклу становить 392716 машинних циклів. Із

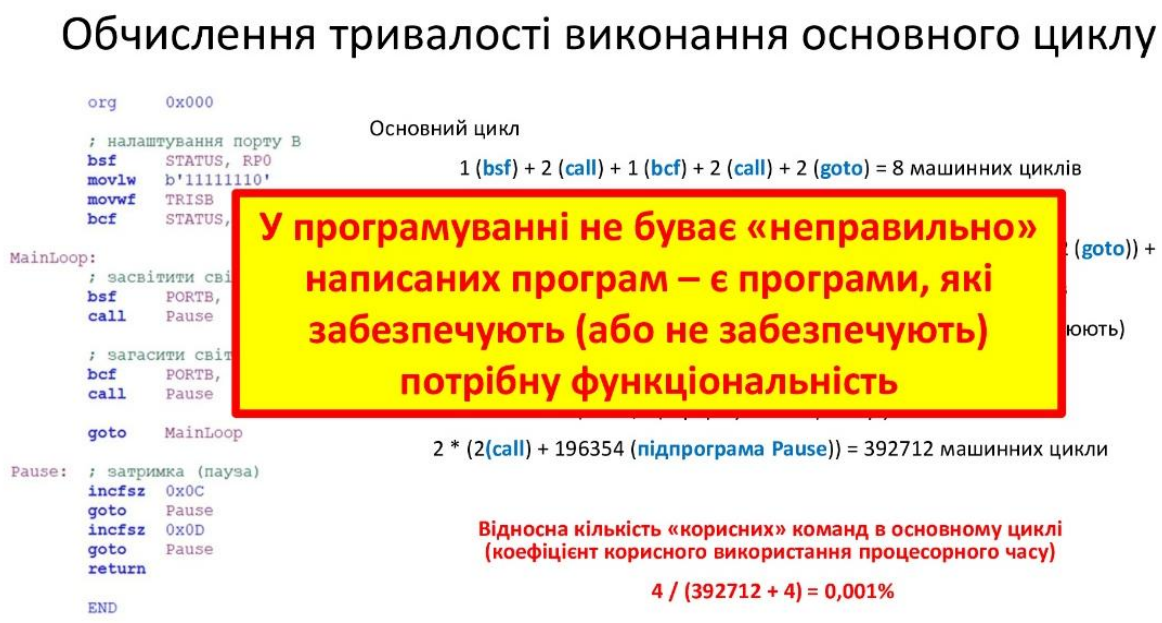
них лише чотири машинних цикли припадають на корисні команди, а весь інший час витрачається на виконання допоміжних операцій, що не несуть прямої користі.

У результаті коефіцієнт корисного використання процесорного часу становить приблизно 0,001%. Саме такі параметри має програма, написана на першому занятті, що і підводить до питання: чи доцільно використовувати саме такий підхід?

І так, і ні. У програмуванні немає програм, які працюють «неправильно». Ви або забезпечуєте функціональність, визначену технічним завданням, або ні. У першому варіанті програми «Вітаю, світе!» у технічному завданні вимагалося, щоб світлодіод мерехтів із заданим інтервалом. Ця вимога виконана, отже необхідна функціональність реалізована.

З точки зору здорового глузду все зроблено правильно. Проте якщо, окрім мерехтіння світлодіода, потрібно буде виконувати додаткові операції, виникають проблеми. Увесь процесорний час використовується лише для формування цього мерехтіння, і для інших задач ресурсів уже не залишається. У такому випадку ефективність роботи системи є низькою.

Водночас, якщо технічне завдання виконане, зазвичай нікого не цікавить, яким саме способом це було реалізовано. Однак після цього логічно виникає наступне питання: як підвищити коефіцієнт використання процесорного часу?



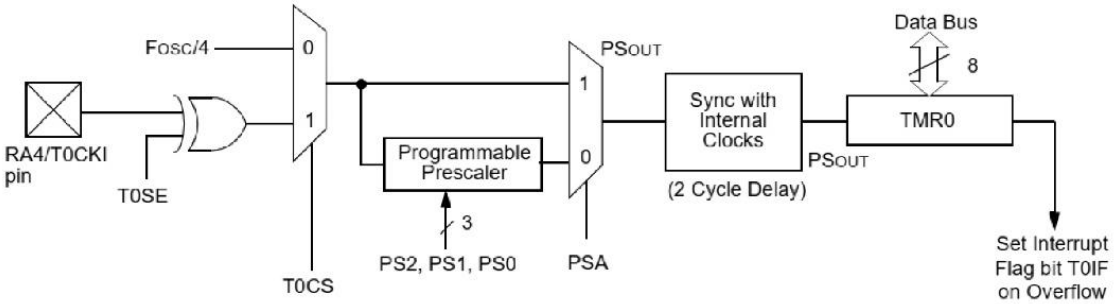
Оскільки мова йде про вимірювання часу, першим апаратним модулем, який необхідно розглянути, є таймери. Відповідно до своєї назви, вони спеціально призначені для вимірювання часових інтервалів. Насправді ці модулі часто називають таймерами-лічильниками, але далі розглядатиметься саме таймер.

У мікроконтролері PIC16F84 присутній лише один таймер. У більш сучасних мікроконтролерах таймерів може бути декілька, і кожен з них має додаткові режими та функції. Усі ці можливості широко використовуються, але в даному випадку вони не розглядаються, оскільки мова йде про найпростіший варіант.

Таймер мікроконтролера PIC16F84 є одним із найпростіших апаратних таймерів. Його структурна схема наведена на слайді. Цей таймер може виконувати лише базову функцію – підрахунок кількості імпульсів.

Таймер може працювати у двох режимах: підраховувати тактові імпульси внутрішнього тактового генератора або рахувати зовнішні імпульси, що надходять на відповідний вхід мікроконтролера. Саме з цього простого таймера і починається розгляд апаратних засобів вимірювання часу.

Структурна схема таймера мікроконтролера PIC16F84A



Note 1: T0CS, T0SE, PSA, PS2:PS0 (OPTION_REG<5:0>).
2: The prescaler is shared with Watchdog Timer (refer to Figure 5-2 for detailed block diagram).

Головним елементом таймера є регістр, який називається TMR0. Це регістр оперативної пам'яті, який може апаратно змінюватися з певною частотою.

Оперативна пам'ять мікроконтролера PIC16F84 поділяється на регістри загального призначення та регістри спеціального призначення. Регістр TMR0 належить до регістрів спеціального призначення і розташований за адресою 0x01 у шістнадцятковому форматі. Саме цей регістр є основою роботи таймера.

З точки зору програміста регістр TMR0 доступний для читання і запису, з ним можна виконувати стандартні операції. Водночас необхідно враховувати, що значення цього регістра може змінюватися апаратно, незалежно від виконання програмного коду. Саме тому TMR0 є ключовим елементом таймера.

Регістр TMR0

FIGURE 2-2: REGISTER FILE MAP - PIC16F84A

File Address			File Address
00h	Indirect addr.(1)	Indirect addr.(1)	80h
01h	TMR0	OPTION_REG	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	—	—	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2(1)	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch			8Ch

- Основний регістр таймера TMR0 є частиною оперативної пам'яті (регістр) спеціального призначення
- Регістр TMR0 програмно доступний для читання та запису
- **Зміст регістру TMR0 збільшується апаратно**

Тепер виникає питання, яким чином відбувається збільшення значення таймера. Існує кілька варіантів, тобто кілька джерел імпульсів, які можуть призводити до інкрементування регістра таймера.

Перший варіант, який буде основним у подальших практичних роботах, полягає у використанні сигналу тактового генератора як джерела інкрементування таймера. У цьому режимі на таймер подається не безпосередньо тактова частота, а тактова частота, поділена на чотири. Це зроблено з тієї причини, що одна асемблерна інструкція вважається такою, що виконується за чотири такти, тобто за один машинний цикл.

У результаті за кожен машинний цикл значення таймера збільшується на одиницю. Це є логічним, оскільки стан мікроконтролера змінюється саме з періодом машинного циклу.

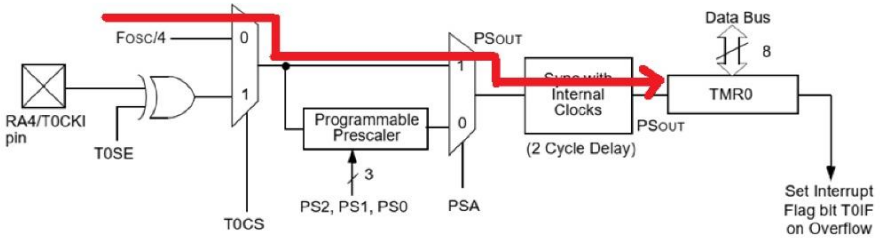
Якби на таймер подавалася тактова частота без поділу на чотири, то за один машинний цикл таймер збільшувався б на чотири, і програмно відстежувати його значення було б практично неможливо.

Таким чином, у цьому режимі регістр TMR0 може збільшуватися кожен машинний цикл. Увімкнення цього режиму здійснюється шляхом встановлення біта T0CS у нуль. Цей біт керує мультиплексером, який вибирає джерело лічильних імпульсів для таймера.

Мультиплексер є вузлом з кількома входами та одним виходом і фактично виконує функцію комутатора. Залежно від значення біта T0CS до виходу підключається відповідне джерело імпульсів. Якщо T0CS дорівнює нулю, джерелом є тактовий генератор із частотою, поділеною на чотири. Якщо T0CS дорівнює одиниці, таймер починає рахувати зовнішні імпульси.

Зовнішні імпульсні сигнали в цьому випадку подаються на вивід RB4, який може використовуватися як джерело імпульсів для інкрементування таймера.

Структурна схема таймера мікроконтролера PIC16F84A



Note 1: T0CS, T0SE, PSA, PS2:PS0 (OPTION_REG<5:0>).
2: The prescaler is shared with Watchdog Timer (refer to Figure 5-2 for detailed block diagram).

Якщо біт T0CS = 0, а біт PSA = 1, то регістр TMR0 буде збільшуватися на одиницю у кожному машинному циклі

Виникає питання, де саме розташовані ці біти. Вони знаходяться в оперативній пам’яті, у регістрі спеціального призначення OPTION_REG.

Регістр OPTION_REG є регістром налаштувань таймера. Це бітова мапа, у якій кожен окремий біт відповідає за певний параметр роботи таймера та пов’язаних з ним вузлів.

Для того щоб зрозуміти, який саме біт за що відповідає, необхідно звертатися до технічної документації на мікроконтролер. Саме в ній наведено повний опис регістра OPTION_REG і призначення кожного його біта.

Регістр OPTION_REG

FIGURE 2-2: REGISTER FILE MAP - PIC16F84A

File Address			File Address
00h	Indirect addr.(1)	Indirect addr.(1)	80h
01h	TMR0	OPTION_REG	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	—	—	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2(1)	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch			8Ch

REGISTER 2-2: OPTION REGISTER (ADDRESS 81h)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **RBPU**: PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI pin
0 = Increment on low-to-high transition on RA4/T0CKI pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

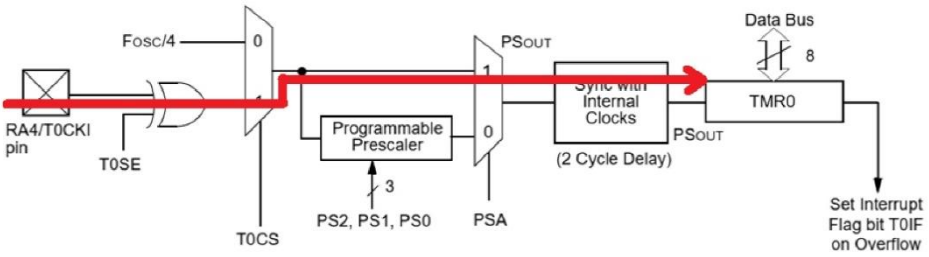
bit 2-0 **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Отже, якщо біт T0CS встановлений у нуль, джерелом тактування таймера є тактовий генератор мікроконтролера з частотою, поділеною на чотири. У цьому режимі таймер інкрементується кожен машинний цикл.

Якщо біт T0CS встановлений в одиницю, джерелом тактових імпульсів для таймера стає зовнішній сигнал, який подається на вивід RB4. У цьому випадку таймер працює як лічильник зовнішніх імпульсів.

Структурна схема таймера мікроконтролера PIC16F84A



Note 1: T0CS, T0SE, PSA, PS2:PS0 (OPTION_REG<5:0>).
2: The prescaler is shared with Watchdog Timer (refer to Figure 5-2 for detailed block diagram).

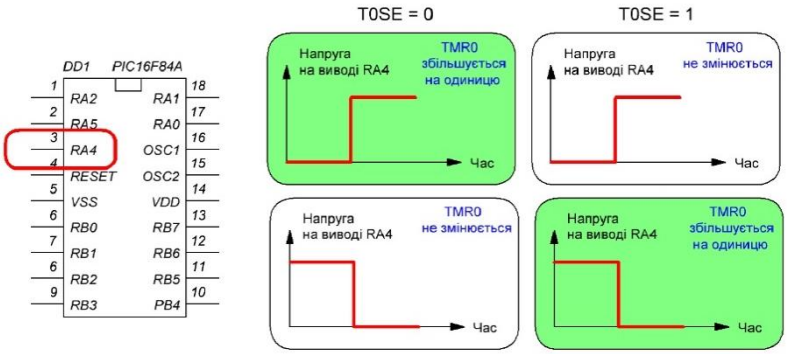
Якщо біт T0CS = 1, а біт PSA = 1, то регістр TMR0 буде збільшуватися на одиницю при зміні рівня сигналу на виводі RA4

Зовнішній сигнал повинен бути імпульсним. За допомогою біта T0SE визначається, за яким фронтом цього сигналу буде збільшуватися регістр TMR0. Якщо інкремент має відбуватися при переході сигналу на виводі RB4 з нуля в одиницю, біт T0SE встановлюється в нуль. Якщо ж інкремент повинен відбуватися при переході сигналу з одиниці в нуль, біт T0SE необхідно встановити в одиницю.

Вибір цього режиму залежить від конкретної специфіки застосунку. На даному етапі такі складні режими не використовуються. У практичних роботах таймер буде тактуватися від внутрішнього тактового генератора, тобто біт T0CS встановлюється в нуль.

Біт T0SE також розташований у регістрі спеціального призначення OPTION_REG, і його значення визначається програмно під час налаштування таймера.

Вибір фронту/спаду імпульсу



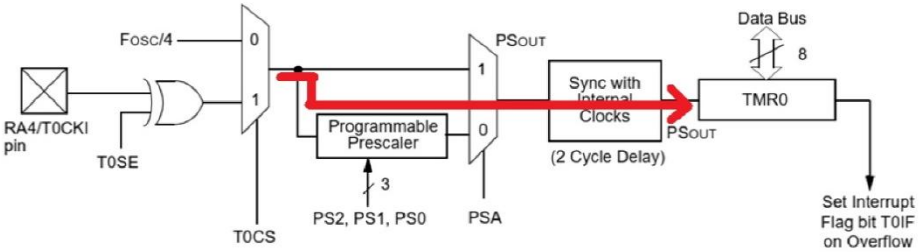
Другим бітом, який також необхідно налаштовувати, є біт PSA. Біт PSA визначає, чи буде джерело тактування подаватися на регістр таймера безпосередньо, чи між джерелом імпульсів і таймером буде використаний передподільник частоти, який зазвичай називають прескалером.

Прескалер є спеціальним апаратним вузлом, що ділить частоту вхідних імпульсів на певний коефіцієнт, як правило шляхом послідовного ділення на два. У мікроконтролері PIC16F84 прескалер є спільним ресурсом і використовується або таймером загального призначення TMR0, або сторожовим таймером.

Сторожовий таймер розглядався раніше як апаратний вузол, що примусово перезавантажує мікроконтролер у разі некоректної роботи програмного забезпечення. Для нього також існують власні налаштування, і саме тому прескалер не може одночасно обслуговувати обидва таймери.

Біт PSA визначає, з яким саме вузлом буде працювати передподільник частоти. Якщо біт PSA встановлений в одиницю, прескалер не використовується таймером TMR0 і залишається прив'язаним до сторожового таймера. Якщо ж біт PSA встановлений у нуль, прескалер підключається до таймера загального призначення TMR0 і використовується для поділу його тактових імпульсів.

Структурна схема таймера мікроконтролера PIC16F84A



Note 1: T0CS, T0SE, PSA, PS2:PS0 (OPTION_REG<5:0>).
2: The prescaler is shared with Watchdog Timer (refer to Figure 5-2 for detailed block diagram).

Якщо біт PSA = 0, то регістр TMR0 буде підключений до джерела тактового сигналу через програмований передподільник частоти

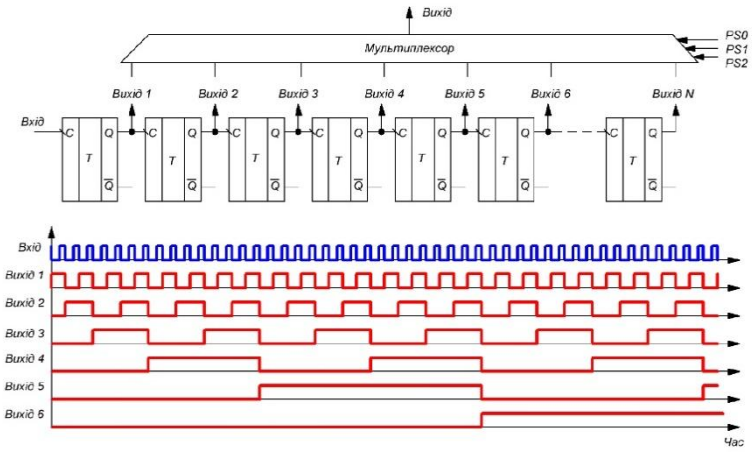
Передподільник частоти побудований таким чином. Зверніть увагу на слово «програмований», тобто його коефіцієнт передачі або коефіцієнт ділення можна змінювати програмно. Апаратно передподільник частоти складається з кількох Т-тригерів, з'єднаних послідовно, тобто вхід Т-тригера наступної ланки підключений до виходу Т-тригера попередньої.

Т-тригер – це апаратний вузол, який змінює свій стан при зміні тактового сигналу. Він перемикається у протилежний стан при зміні вхідного сигналу, наприклад, з одиниці в нуль, як це показано на схемі. Один такий Т-тригер ділить частоту вхідних імпульсів на два.

Коли такі Т-тригери з'єднані послідовно, на виході першого ми маємо частоту, поділену на 2, на виході другого – уже поділену на 4, на виході третього – на 8, далі на 16, 32 і так далі. До виходів усіх цих тригерів підключений один великий мультиплексор з вісьмома входами і одним виходом.

Біти PS0, PS1 та PS2 якраз і визначають, до якого саме входу мультиплексора буде підключений його вихід, тобто який коефіцієнт поділу частоти буде використаний у конкретний момент часу.

Передподільник частоти

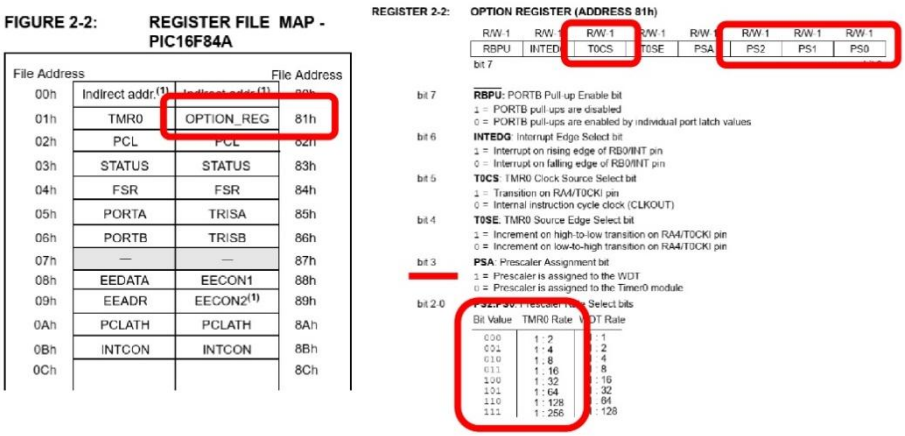


Біти **PS0 – PS3** регістру **INTCON** визначають який з виходів перед-подільника буде активним

І це дозволяє нам шляхом встановлення бітів PS0, PS1 та PS2 обирати коефіцієнт передачі передподільника. Тобто тактова частота, яка вже поділена на чотири, може бути додатково поділена ще кілька разів.

У граничному випадку ми можемо поділити її до коефіцієнта 1:256. Це відбувається тоді, коли всі біти PS0, PS1 та PS2 встановлені в значення, що дорівнює одиниці.

Регістр OPTION_REG



Ну і, власне кажучи, навіщо нам усе це? Якщо розглядати лише те, що ми маємо зараз, то виходить, що у нас є певна комірка оперативної пам’яті, яка апаратно збільшується з деякою частотою. Збільшується й збільшується – на перший погляд, нічого особливого, адже під час виконання програми оперативна пам’ять і так постійно змінюється.

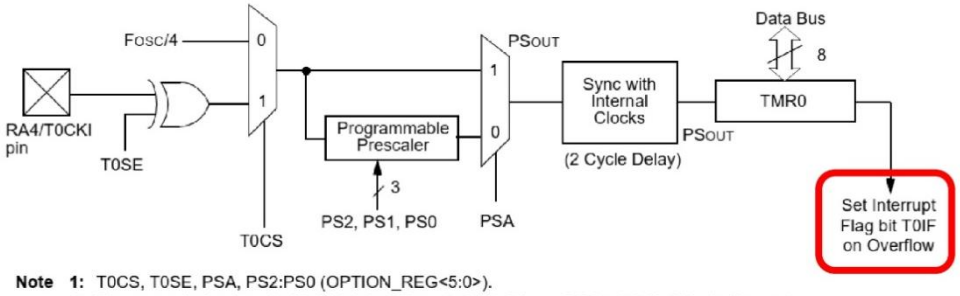
Головне ж полягає в іншому. Таймер спроможний генерувати переривання. Регістр TMR0 поступово збільшується: після перезавантаження він дорівнює нулю, потім через певний час стає одиницею, двійкою, трійкою і так далі, аж поки не доходить до 255.

Далі відбувається наступне. Коли значення 255 збільшується ще на одиницю, у восьмирозрядній системі це дає нуль. Виникає переповнення таймера – перехід зі значення 255 у нуль. Саме в цей момент і формується переривання.

Отже, основне призначення таймера полягає в генерації апаратних переривань через певні інтервали часу. Усі ці налаштування – вибір джерела тактування, коефіцієнтів поділу, передподільника – потрібні лише для того, щоб переривання нашого програмного коду виникали з заданою періодичністю, прив’язаною до тактового генератора.

У більшості практичних випадків як джерело тактування для таймера використовується саме внутрішній тактовий генератор мікроконтролера, і саме за такою схемою таймер найчастіше й застосовують.

Структурна схема таймера мікроконтролера PIC16F84A



Note 1: T0CS, T0SE, PSA, PS2:PS0 (OPTION_REG<5:0>).
2: The prescaler is shared with Watchdog Timer (refer to Figure 5-2 for detailed block diagram).

При кожному переповненні регістру TMR0 (зміні 255 → 0) може виникнути переривання

Що відбувається, коли у нас виникає переривання? Перш за все потрібно зрозуміти, де ми можемо це переривання побачити або відслідкувати.

У випадку таймера момент переривання відповідає переповненню. Коли значення регістру TMR0 переходить з 255 у нуль, апаратно встановлюється біт T0IF у регістрі INTCON.

Регістр INTCON – це скорочення від Interrupt Control. Біт T0IF є прапором переривання переповнення таймера. Він встановлюється апаратно в будь-якому випадку, незалежно від того, чи дозволені переривання від таймера, чи ні.

Якщо регістр TMR0 апаратно збільшується, то в момент переходу його значення з 255 у нуль відповідний прапор T0IF обов'язково встановлюється. Такі біти називають прапорами переривань, оскільки при виникненні переривання одразу піднімається прапор відповідного джерела.

Регістр INTCON
(Interrupt configuration)

FIGURE 2-2: REGISTER FILE MAP - PIC16F84A

File Address			File Address
00h	Indirect addr.(1)	Indirect addr.(1)	80h
01h	TMR0	OPTION_REG	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	—	—	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2(1)	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch			8Ch

REGISTER 2-3: INTCON REGISTER (ADDRESS 0Bh, 8Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 **EEIE**: EE Write Complete Interrupt Enable bit
1 = Enables the EE Write Complete interrupts
0 = Disables the EE Write Complete interrupt

bit 5 **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3 **RBIE**: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **T0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

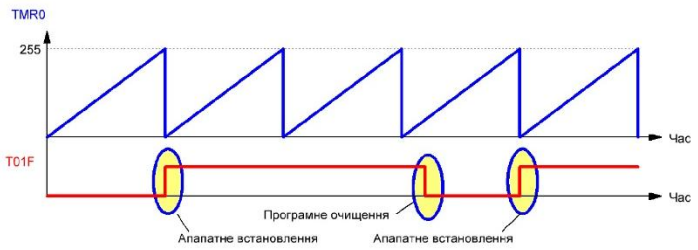
bit 1 **INTF**: RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0 **RBIF**: RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Прапор переривання скидається програмним шляхом. Програміст може встановлювати цей біт як у нуль, так і в одиницю. В одиницю він може встановлюватися як програмно, так і апаратно, а скидання в нуль можливе лише програмно. Для цього програміст повинен явно записати відповідну інструкцію в коді.

Момент скидання прапора переривання повністю визначається програмою. Апаратно цей прапор встановлюється автоматично без участі програміста при кожному переповненні таймера, тобто при переході значення регістру TMR0 з 255 у 0. Таким чином, при переповненні таймера прапор переповнення піднімається, але саме переривання ще не виникає.

Біт (прапор) T01F



Прапор T01F встановлюється апаратно при переповненні регістру TMR0. Його також можна встановити або очистити програмно (наприклад, командами bsf або bcf)

Переривання виникає лише за умови встановлення в регістрі кількох керуючих бітів. Для дозволу переривання від таймера необхідно програмно встановити біт T0IE, який відповідає за дозвіл переривань переповнення таймера, а також біт GIE, що забезпечує

глобальну активізацію всіх переривань у мікроконтролері. Біт GIE або дозволяє, або забороняє всі переривання одночасно, тоді як біти з суфіксом IE керують перериваннями від окремих апаратних вузлів. У даному випадку використовується біт T0IE.

Фактичне виникнення переривання можливе лише тоді, коли додатково встановлений прапор T0IF. Цей прапор піднімається апаратно при переповненні таймера і програмно не встановлюється. Таким чином, переривання виникає лише за одночасної наявності встановлених бітів GIE, T0IE та T0IF. Біти GIE і T0IE повинні бути встановлені програмно для дозволу переривання від таймера, тоді як T0IF формується автоматично апаратурою.

Усі зазначені біти розміщені в одному регістрі керування, в якому також знаходяться біти дозволу та прапори інших джерел переривань, що дозволяє аналогічним чином керувати й іншими перериваннями мікроконтролера.

Регістр INTCON
(Interrupt configuration)

FIGURE 2-2: REGISTER FILE MAP - PIC16F84A

File Address			File Address
00h	Indirect addr.(1)	Indirect addr.(1)	80h
01h	TMR0	OPTION_REG	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	—	—	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2(1)	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch			8Ch

REGISTER 2-3: INTCON REGISTER (ADDRESS 0Bh, 8Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 **EEIE:** EE Write Complete Interrupt Enable bit
1 = Enables the EE Write Complete interrupts
0 = Disables the EE Write Complete interrupt

bit 5 **T0IE:** TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE:** RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

Для того щоб виникло переривання необхідно встановити біти T0IE – (Timer 0 Interrupt Enable) та GIE (Global Interrupt Enable)

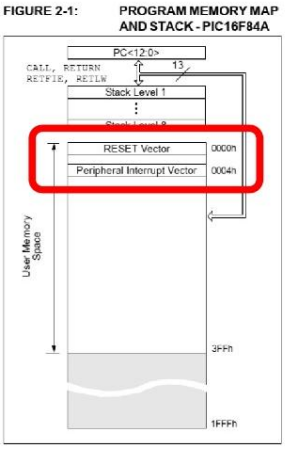
Для налаштування переривань використовується регістр INTCON, який є основним регістром конфігурації переривань. Саме в ньому зосереджені біти глобального дозволу переривань, біти дозволу окремих джерел і відповідні прапори. Після того як усі необхідні умови виконані і біти GIE, T0IE та T0IF встановлені в одиницю, виникає апаратне переривання.

У момент виникнення переривання ядро мікроконтролера зберігає в стек адресу наступної команди, тобто адресу поточної виконуваної команди, збільшену на одиницю. Таким чином запам’ятовується місце, куди необхідно буде повернутися після завершення обробки переривання. Після цього мікроконтролер апаратно переходить до виконання команди, що знаходиться в пам’яті програм за фіксованою адресою 0x0004. Ця комірка є вектором переривання, аналогічно до того, як комірка з адресою 0x0000 є вектором скидання, з якого починається виконання програми після перезавантаження.

Одночасно з переходом до вектора переривання апаратно очищується біт GIE, що забороняє всі переривання на час обробки поточного. Це необхідно для того, щоб під час виконання обробника не виникали нові переривання і не відбувався повторний вхід у той самий обробник. Далі виконуються інструкції обробника переривання, які програміст задає самостійно.

Виконання обробника триває до зустрічі команди RETFIE. Після виконання цієї команди відбувається відновлення нормального режиму роботи: апаратно встановлюється біт GIE, зі стека витягується збережена адреса, і ядро мікроконтролера повертається до виконання основної програми, починаючи з команди, наступної за тією, під час виконання якої виникло переривання. Таким є загальний порядок обробки переривань.

Порядок обробки переривань мікроконтролером



- Запам'ятовується адреса поточної команди, яка повинна була виконуватись у момент виникнення переривання (використовується стек)
- Починається виконуватися команда, розташована на векторі переривання (Peripheral Interrupt Vector) (адреса 0x0004 у пам'яті програм)
- Апаратно забороняється усі переривання (біт **GIE** очищується і може бути встановлений програмно)
- Виконуються команди допоки не зустрінеться команда **retfie** (Return from interrupt)
- Після цього біт **GIE** апаратно встановлюється і починає виконуватися команда, яка повинна була виконуватися у момент виникнення переривання (при цьому звільняється одна комірка стеку)

Команда RETFIE не виконує жодних обчислювальних дій і не має параметрів, що впливають на логіку програми. Її єдине призначення полягає у завершенні обробки переривання і поверненні до основного виконання програми. У межах обробника переривань така команда може зустрічатися кілька разів, зокрема у випадках, коли алгоритм обробки має розгалужену структуру і вихід з обробника можливий з різних ділянок коду.

Команда **retfie**

RETFIE	**Return from Interrupt**
Syntax:	[*label*] RETFIE
Operands:	None
Operation:	TOS → PC, 1 → GIE
Status Affected:	None

- Команда **retfie** – спеціалізована команда для апаратного завершення переривання
- В обробнику переривань може бути декілька команд **retfie** (для пришвидшення виконання розгалужених алгоритмів
- Якщо в обробнику переривань не очищений хоча б один прапор, що викликає переривання, обробник переривання буде викликаний знову

Виникає проблема з розміщенням програми в пам'яті програм. Раніше програма «Вітаю, світе!» розташовувалась, починаючи з вектора перезавантаження, тобто з адреси 0x0000, далі 0x0001, 0x0002, 0x0003 і так далі. У такому випадку комірка пам'яті програм з адресою 0x0004 вже містить певну асемблерну команду.

Адреса 0x0004 є вектором переривань, і при виникненні переривання мікроконтролер починає виконання саме з цієї комірки. У наведеному прикладі так співпало, що за адресою 0x0004 знаходиться початок основного циклу, але це не є правилом – там могла бути будь-яка команда з будь-якого блоку, зокрема з ініціалізації чи іншої частини програми. Суть у тому, що вектор переривань уже зайнятий кодом основної програми, і це потрібно виправити.

Для цього необхідно перерозподілити розміщення програми в пам'яті програм так, щоб область вектора переривань використовувалась за призначенням.

Особливості розташування коду у пам’яті програм

```
        org     0x000

        ; налаштування порту B
        bsf     STATUS, RP0
        movlw   b'11111110'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
        ; засвітити світлодіод
        bsf     PORTB, RB0
        call    Pause

        ; загасити світлодіод
        bcf     PORTB, RB0
        call    Pause

        goto    MainLoop

Pause:  ; затримка (пауза)
        incfsz  0x0C
        goto    Pause
        incfsz  0x0D
        goto    Pause
        return

        END
```

Адреса	Мітка	Команда
0x0000		bsf STATUS, RP0
0x0001		movlw b'11111110'
0x0002		movwf TRISB
0x0003		bcf STATUS, RP0
0x0004	MainLoop:	bsf PORTB, RB0
0x0005		call Pause
0x0006		bcf PORTB, RB0
0x0007		call Pause
0x0008		goto MainLoop
0x0009	Pause:	incfsz 0x0C, f
0x000A		goto Pause
0x000B		incfsz 0x0D, f
0x000C		goto Pause
0x000D		return

Обробка переривання почнеться з команди, яка розташована у цій комірці

Перерозподіл розміщення коду в пам’яті програм виконується директивою компілятора **org**. Раніше на початку програми задавали **org** 0x0000, щоб наступна команда розташовувалась у комірці пам’яті програм з нульовою адресою. Тепер потрібно використати дві директиви **org**: першу – **org** 0x0000 для вектора перезавантаження, другу – **org** 0x0004 для коду, який має виконуватися при виникненні переривання.

Між вектором перезавантаження і вектором переривань міститься лише три комірки, тому в цьому діапазоні не розміщують корисний код. У комірці за адресою 0x00 розміщують одну команду **goto init**, де **init** – мітка блоку ініціалізації, який тепер розташовується після обробника переривань. Комірки між 0x0000 і 0x0004 не використовуються для виконання в нормальному режимі.

З адреси 0x0004, тобто з вектора переривань, починається обробник переривань – послідовність команд, що виконується при перериванні. Усередині цього блоку обов’язково має бути щонайменше одна команда RETFIE, інакше мікроконтролер не зможе вийти зі стану обробки переривання. Найчастіше RETFIE розташовують наприкінці обробника, хоча можливі кілька таких команд для різних гілок обробки.

Після обробника переривань у пам’яті програм розташовується блок ініціалізації, який виконується один раз і містить налаштування мікроконтролера. До цього блоку потрапляють одразу після перезавантаження через **goto init**. Далі розміщується основний нескінченний цикл, а вже після нього можна розташовувати підпрограми, таблиці та інші потрібні дані. Якщо в проєкті використовуються переривання, структура розміщення коду має бути саме такою.

```
        org     0x0000
        goto    Init

        ; обробник переривань
        org     0x0004
        ; якась команда
        ; якась команда
        ; якась команда
        ; якась команда
        ; якась команда
        ; якась команда
        retfie

Init:
        bsf     STATUS, RP0
        movlw   b'11111110'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
        bsf     PORTB, RB0
        call    Pause
        bcf     PORTB, RB0
        call    Pause
        goto    MainLoop

Pause:
        incfsz  0x0C, f
        goto    Loop2
        incfsz  0x0D, f
        goto    Loop2
        return

        END
```

Адреса	Мітка	Команда
0x0000		goto Init
0x0001		
0x0002		
0x0003		
0x0004		Якась команда
0x0005		Якась команда
0x0006		Якась команда
0x0007		Якась команда
0x0008		Якась команда
0x0009		Якась команда
0x000A		retfie
0x000B	Init:	bsf STATUS, RP0
0x000C		movlw b'11111110'
0x000D		movwf TRISB
0x000E		bcf STATUS, RP0
0x000F	MainLoop:	bsf PORTB, RB0
0x0010		call Pause
0x0011		bcf PORTB, RB0
0x0012		call Pause
0x0013		goto MainLoop
0x0014	Pause:	incfsz 0x0C, f
0x0015		goto Pause
0x0016		incfsz 0x0D, f
0x0017		goto Pause
0x0018		return

Ці комірки не використовуються

Обробник переривання

Блок ініціалізації

Основний цикл

Підпрограма

Для коректного розташування коду у пам’яті програм використовується директива **org**

В типовому програмному забезпеченні для мікроконтролерів PIC, що використовують переривання з’являються комірки пам’яті програм, які не можна використати

У мікроконтролері PIC16F84 передбачено чотири джерела переривань. Одним із них є таймер. Іншим джерелом є енергонезалежна пам’ять даних і програм, яка використовується для збереження користувацьких даних. Зчитування з цієї пам’яті відбувається швидко, тоді як операція запису займає певний час, тому для уникнення очікування завершення запису може використовуватися механізм переривань.

Джерелами переривань також можуть бути порти введення-виведення RB4–RB7. Це зручно, наприклад, при реалізації клавіатури, коли до цих виводів підключаються кнопки. При натисканні кнопки змінюється рівень сигналу на відповідному виводі, що призводить до виникнення переривання, яке далі обробляється програмно.

Окремим джерелом переривання є канал RB0. Його можна конфігурувати так, щоб переривання виникало за фронтом або за спадом імпульсу. На відміну від RB4–RB7, де переривання формується при будь-якій зміні сигналу, канал RB0 виконує попередню обробку і фіксує саме ту подію, яка задана під час конфігурації.

Джерела переривань у мікроконтролері PIC16F84A

6.8 Interrupts

The PIC16F84A has 4 sources of interrupt:

- External interrupt RB0/INT pin
- TMR0 overflow interrupt
- PORTB change interrupts (pins RB7:RB4)
- Data EEPROM write complete interrupt

- Зовнішнє переривання на виводі RB0
- Переповнення таймеру TMR0
- Зміна стану сигналів на виводах RB4 – RB7
- Закінчення запису у внутрішню енергонезалежну пам’ять EEPROM

У мікроконтролері PIC16F84 наявні чотири джерела переривань, але всі вони приводять до одного обробника, оскільки існує лише одна комірка пам’яті програм, яка є вектором переривань. У більш розвинених мікроконтролерах таких векторів може бути кілька, що спрощує організацію обробників, але в даному випадку використовується єдиний вектор для всіх джерел.

Якщо активоване лише одне джерело переривання, наприклад таймер, то при виникненні переривання програміст однозначно знає його причину. У разі, коли дозволено кілька джерел переривань, необхідно визначити, яке саме з них спрацювало. Це робиться шляхом перевірки відповідних прапорів переривань.

З кожним джерелом переривання пов’язані два біти. Один з них є біт дозволу, який програмно вмикає або вимикає відповідне переривання. Другий є прапором переривання, що сигналізує про факт виникнення події. Біти дозволу встановлюються і очищуються програмно, прапори переривань встановлюються апаратно, а скидаються програмно. У PIC16F84 це реалізовано саме таким чином, хоча в інших мікроконтролерах можливі відмінності, тому для коректної роботи необхідно уважно вивчати технічну документацію щодо механізму скидання прапорів переривань.

Як дізнатися яке переривання спрацювало?

REGISTER 3-1: EECON1 REGISTER (ADDRESS 88h)

U-0	U-0	U-0	R/W-0	R/W-x	R/W-0	R/S-0	R/S-0
—	—	—	EEIF	WRERR	WREN	WR	RD
bit 7							bit 0

bit 7-5 **Unimplemented:** Read as '0'

bit 4 **EEIF:** EEPROM Write Operation Interrupt Flag bit
1 = The write operation completed (must be cleared in software)
0 = The write operation is not complete or has not been started

bit 3 **WRERR:** EEPROM Error Flag bit
1 = A write operation is prematurely terminated (any MCLR Reset or any WDT Reset during normal operation)
0 = The write operation completed

bit 2 **WREN:** EEPROM Write Enable bit
1 = Allows write cycles
0 = Inhibits write to the EEPROM

REGISTER 2-3: INTCON REGISTER (ADDRESS 0Bh, 8Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 **EEIE:** EE Write Complete Interrupt Enable bit
1 = Enables the EE Write Complete interrupts
0 = Disables the EE Write Complete interrupt

bit 5 **T0IE:** TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE:** RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3 **RBIE:** RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **T0IF:** TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF:** RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0 **RBIF:** RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Дізнатися яке переривання спрацювало можна шляхом перевірки відповідних прапорів

До переваг мікроконтролера PIC16F84 належить простий механізм виходу з обробника переривань, оскільки прапори переривань скидаються програмно. Активізація переривань здійснюється за допомогою бітів дозволу.

Усього використовується п’ять бітів enable: чотири з них відповідають конкретним джерелам переривань, а біт GIE виконує роль глобального дозволу і вмикає або вимикає всі переривання одночасно. Усі ці біти зосереджені в регістрі INTCON.

Як активізувати переривання?

REGISTER 2-3: INTCON REGISTER (ADDRESS 0Bh, 8Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

bit 7 **GIE**: Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 **EEIE**: EE Write Complete Interrupt Enable bit
1 = Enables the EE Write Complete interrupts
0 = Disables the EE Write Complete interrupt

bit 5 **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3 **RBIE**: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **T0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF**: RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0 **RBIF**: RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Активізувати переривання можна шляхом встановлення відповідних біт у регістрі INTCON

Усі переривання (разом) можна заборонити або дозволити шляхом встановлення біту GIE у регістрі INTCON

Існує рекомендований порядок обробки переривань, який може змінюватися залежно від конкретної задачі, але зазвичай дотримується типової послідовності. На початку обробника необхідно зберегти контекст виконання. Після цього, якщо дозволено кілька джерел переривань, перевіряються прапори для визначення причини переривання; у разі використання лише одного джерела цей етап може бути пропущений.

Далі виконується код, що безпосередньо реалізує потрібну реакцію на переривання. Після завершення цієї частини обов’язково скидається відповідний прапор переривання. Якщо прапор не буде скинутий, після виходу з обробника переривання одразу виникне знову, оскільки апаратна умова його генерації залишиться активною. Для коректного завершення обробки необхідно привести апаратний стан у норму шляхом очищення відповідного прапора. Після цього відновлюється збережений контекст і здійснюється вихід з обробника переривання.

Порядок обробки переривань

- Зберегти контекст
- Перевірити прапор (якщо використовуються кілька джерел переривань)
- Виконати код, що відповідає за обробку даного переривання
- **Очистити прапор відповідного переривання**
- Відновити контекст
- Вийти з переривання

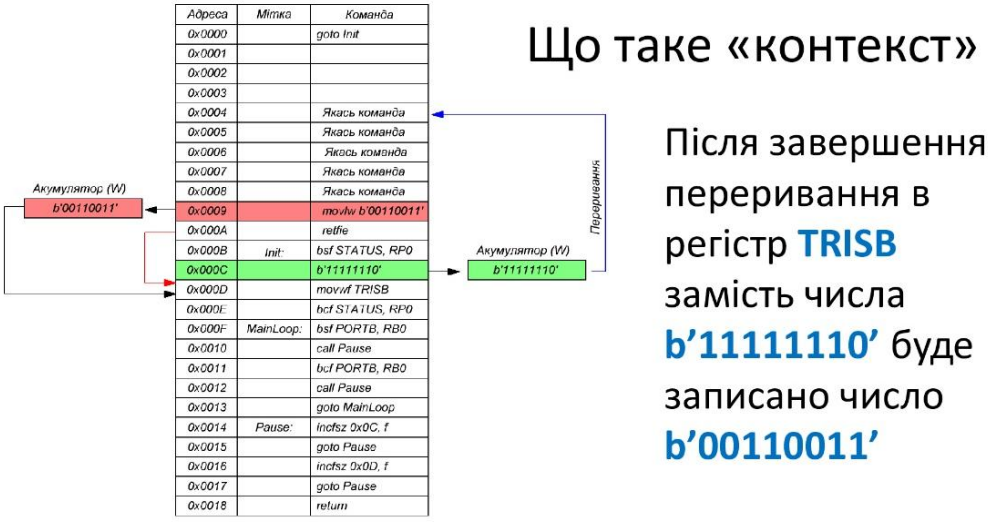
Якщо не очистити прапор, що спричинив виклик переривання, то після виходу із переривання його обробник буде викликаний знову!

Контекст – це стан процесора в момент виконання основної програми. Розглянемо ситуацію на прикладі простої програми. Нехай у процесі виконання програма дійшла до команди, що розташована за адресою 0x000C. Команди з адрес 0x000C і 0x000D використовуються для завантаження певного значення в регістр 0x003B. Це значення спочатку завантажується в акумулятор командою MOVLW, а потім копіюється з акумулятора в регістр командою MOVWF, оскільки безпосередньо завантажити число в регістр неможливо.

У цей момент між двома командами або після завантаження значення в акумулятор може виникнути переривання. Основний цикл переривається, і виконання переходить до коду за адресою 0x0004, тобто до обробника переривань. У процесі обробки переривання в акумулятор завантажується інше значення, необхідне для роботи обробника, оскільки акумулятор є універсальним регістром і використовується практично в усіх операціях.

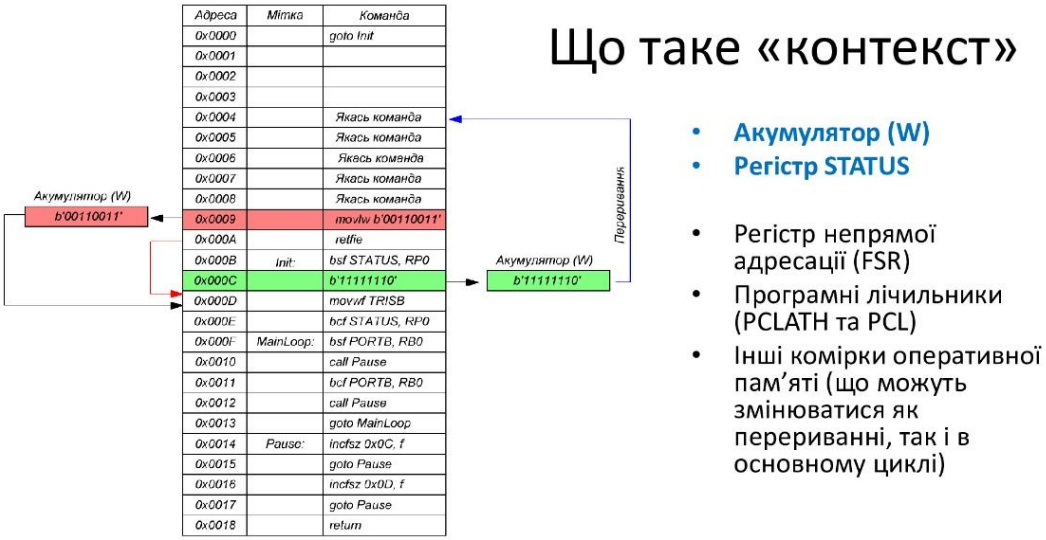
Після виконання команди RETFIE відбувається повернення з переривання, і процесор продовжує виконання основної програми з команди, наступної за тією, на якій виникло переривання. У даному випадку це команда запису з акумулятора в регістр 0x003B. Проте в акумуляторі вже знаходиться не те значення, яке передбачалося під час написання коду, а інше, змінене в обробнику переривань. У результаті в регістр записується неправильне значення, і програма починає працювати некоректно.

Саме тому перед початком обробки переривання необхідно зберігати контекст. Контекстом у цьому випадку є вміст регістрів, зокрема акумулятора, який може бути змінений під час обробки переривання.



Які саме дані необхідно зберігати, залежить від конкретної задачі та структури програми. Найчастіше зберігають два регістри – акумулятор і регістр статусу спеціального призначення, оскільки вони безпосередньо впливають на виконання подальших інструкцій.

У разі використання більш складних методів програмування може виникнути потреба зберігати регістр непрямої адресації, програмний лічильник або окремі комірки оперативної пам’яті. Чи потрібно це робити, визначається логікою конкретної програми. Саме сукупність збережених значень регістрів і даних і становить контекст.



Вміст акумулятора і регістра статусу не можна просто скопіювати в оперативну пам’ять і потім відновити довільними командами. Тому в документації на мікроконтролер наведений рекомендований код збереження контексту, в якому використовується спеціальний порядок дій і команда SWAP.

Збереження контексту починається з акумулятора, оскільки всі подальші операції виконуються саме через нього. Акумулятор зберігається першим і відновлюється останнім. Після цього зберігається регістр статусу. Для цього його вміст спочатку завантажується в акумулятор, але не безпосередньо, а з використанням команди SWAP, яка змінює порядок напівбайтів. Далі вміст акумулятора записується в окрему комірку оперативної пам’яті, призначену для збереження регістра статусу.

Після завершення обробки переривання відновлення контексту виконується у зворотному порядку, що забезпечує коректне повернення стану процесора до того, який був до виникнення переривання.

Рекомендований код збереження контексту

EXAMPLE 6-1: SAVING STATUS AND W REGISTERS IN RAM

```
PUSH    MOVWF   W_TEMP            ; Copy W to TEMP register,
        SWAPF   STATUS,     W     ; Swap status to be saved into W
        MOVWF   STATUS_TEMP       ; Save status to STATUS_TEMP register
ISR     :                         :
        :                         ; Interrupt Service Routine
        :                         ; should configure Bank as required
        :                         ;
POP     SWAPF   STATUS_TEMP,W     ; Swap nibbles in STATUS_TEMP register
                                  ; and place result into W
        MOVWF   STATUS            ; Move W into STATUS register
                                  ; (sets bank to original state)
        SWAPF   W_TEMP,     F     ; Swap nibbles in W_TEMP and place result in W_TEMP
        SWAPF   W_TEMP,     W     ; Swap nibbles in W_TEMP and place result into W
```

На початку обробки переривання значення акумулятора (W) зберігається в одній з комірок оперативної пам’яті (W_TEMP), потім зберігається регістр STATUS в іншій комірці (STATUS_TEMP). Перед виходом з переривання спочатку відновлюється регістр STATUS потім акумулятор

Постає питання, чому збереження і відновлення контексту реалізується так складно і чому не можна просто скористатися командами пересилання даних між акумулятором і оперативною пам’яттю. Команди MOVF і MOVWF забезпечують копіювання вмісту між файлом оперативної пам’яті та акумулятором, причому порядок літер визначає напрямок передавання даних: з файлу в акумулятор або з акумулятора у файл.

У рекомендованому коді для роботи з контекстом використовується команда SWAP. Ця команда міняє місцями два півбайти в одному байті. Вона є спадщиною чотирирозрядних мікроконтролерів, які використовувалися раніше, і була збережена для сумісності та можливості перенесення коду. Команда SWAP ділить байт на два півбайти і міняє їх місцями, виконуючи цю операцію з вибраною коміркою оперативної пам’яті.

Саме ця команда застосовується в рекомендованому механізмі збереження і відновлення контексту, оскільки пряме копіювання акумулятора і регістра статусу стандартними командами може призводити до некоректного результату.

Навіщо так складно?

SWAPF	Swap Nibbles in f
Syntax:	[*label*] SWAPF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f<3:0>) → (destination<7:4>), (f<7:4>) → (destination<3:0>)
Status Affected:	None
Description:	The upper and lower nibbles of register 'f' are exchanged. If 'd' is 0, the result is placed in W register. If 'd' is 1, the result is placed in register 'f'.

```
PUSH    MOVWF   W_TEMP
        SWAPF   STATUS,     W
        MOVWF   STATUS_TEMP
ISR     :
        :
        :
        :
POP     SWAPF   STATUS_TEMP,W

        MOVWF   STATUS

        SWAPF   W_TEMP,     F
        SWAPF   W_TEMP,     W
```

1. Зберігаємо зміст акумулятора в комірку W_TEMP
2. Міняємо місцями напівбайти (тетради) в регістрі STATUS, результат зберігаємо в акумуляторі
3. Зберігаємо зміст акумулятора в комірку STATUS_TEMP
4. Оброблюємо переривання
5. Міняємо місцями напівбайти в регістрі STATUS_TEMP, результат зберігаємо в акумуляторі
6. Зберігаємо зміст акумулятора в регістр STATUS
7. Міняємо місцями напівбайти в комірці W_TEMP, результат зберігаємо в комірці W_TEMP
8. Міняємо місцями напівбайти в комірці W_TEMP, результат зберігаємо в акумуляторі

1	1	1	0	0	1	1	1

↓ *swap*

0	1	1	1	1	1	1	0

↓ *swap*

1	1	1	0	0	1	1	1

Чому не використовувати команди mov?

```
movfw   STATUS_TEMP
movwf   STATUS
movfw   W_TEMP
```

Використати звичайні команди пересилання для збереження і відновлення контексту не можна через побічні ефекти, які вони мають. Команди типу MOVF і MOVWF змінюють регістр статусу. Якщо звернутися до технічної документації, видно, що під час виконання таких команд змінюються прапори в регістрі STATUS.

Розглянемо відновлення контексту. Спочатку з оперативної пам’яті в акумулятор завантажується збережений вміст регістра STATUS, після чого командою MOVWF він записується назад у реальний регістр статусу. На цьому етапі регістр STATUS уже відновлений. Далі потрібно відновити акумулятор, для чого використовується команда типу MOVF f, W, яка копіює значення з комірки оперативної пам’яті в акумулятор. Проблема полягає в тому, що ця команда змінює прапор Z у регістрі STATUS. У результаті щойно відновлений регістр статусу одразу ж затирається.

Через це неможливо коректно відновити і акумулятор, і регістр STATUS, використовуючи лише команди пересилання. Саме тому застосовується команда SWAP. Вона міняє місцями два півбайти в байті і при цьому не впливає на регістр статусу. Подвійне використання SWAP не змінює значення байта, оскільки півбайти міняються місцями двічі і повертаються у початковий стан. Завдяки цьому збереження і відновлення контексту виконується коректно без побічної зміни регістра STATUS.

SWAP vs. MOVF

SWAPF	Swap Nibbles in f
Syntax:	[*label*] SWAPF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f<3:0>) → (destination<7:4>), (f<7:4>) → (destination<3:0>)
Status Affected:	None
Description:	The upper and lower nibbles of register 'f' are exchanged. If 'd' is 0, the result is placed in W register. If 'd' is 1, the result is placed in register 'f'.

MOVF	Move f
Syntax:	[*label*] MOVF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f) → (destination)
Status Affected:	Z
Description:	The contents of register f are moved to a destination dependant upon the status of d. If d = 0, destination is W register. If d = 1, the destination is file register f itself. d = 1 is useful to test a file register, since status flag Z is affected.

```
movfw   STATUS_TEMP
movwf   STATUS
movfw   W_TEMP
```

- В наборі асемблерних команд мікроконтролера PIC16F84 немає команди **movfw** (**movfw** = **movf**, w)
- **Команда movf змінює регістр STATUS (прапор Z)**, тому після виконання команди **movfw W_TEMP** зміст регістру **STATUS** буде спотворений

Саме з цієї причини використовується рекомендований алгоритм збереження і відновлення контексту. Практика показує, що не варто ускладнювати рішення і намагатися вигадувати власні підходи, тобто робити «дерев'яний велосипед». Слід застосовувати саме той код збереження контексту, який наведений у технічній документації, оскільки він враховує всі особливості роботи регістрів і команд.

Водночас існує важливий нюанс. Якщо відомо, що в обробнику переривань не змінюється вміст акумулятора або регістра STATUS, тоді збереження і відновлення контексту можна не виконувати. Якщо ж ці регістри змінюються, збереження контексту є обов'язковим, інакше програма працюватиме некоректно, а пошук помилки може зайняти значний час.

Рекомендований код збереження контексту

EXAMPLE 6-1: SAVING STATUS AND W REGISTERS IN RAM

```
PUSH    MOVWF   W_TEMP              ; Copy W to TEMP register,
        SWAPF   STATUS,     W       ; Swap status to be saved into W
        MOVWF   STATUS_TEMP         ; Save status to STATUS_TEMP register
ISR     :                           :
        :                           ; Interrupt Service Routine
        :                           ; should configure Bank as required
        :                           ;
POP     SWAPF   STATUS_TEMP,W       ; Swap nibbles in STATUS_TEMP register
                                    ; and place result into W
        MOVWF   STATUS              ; Move W into STATUS register
                                    ; (sets bank to original state)
        SWAPF   W_TEMP,     F       ; Swap nibbles in W_TEMP and place result in W_TEMP
        SWAPF   W_TEMP,     W       ; Swap nibbles in W_TEMP and place result into W
```

При обробках переривань виконуємо рекомендації розробника, які написані в технічні документації

Якщо в перериваннях акумулятор (W) та регістр STATUS не змінюються, то зберігати контекст не потрібно

У підсумку маємо два варіанти програми «Вітаю світ». Початковий варіант є простим і зрозумілим, він вже був реалізований і коректно працює. Варіант із використанням таймера та переривань є складнішим, кількість команд і загальний обсяг коду в ньому практично вдвічі більші.

Водночас програма з перериваннями використовує процесорний час значно ефективніше. Основний цикл такої програми займає лише незначну частину ресурсів процесора, і приблизно 99,98% процесорного часу залишається доступним для виконання інших корисних задач. Початковий варіант програми використовував увесь процесорний час, через що мікроконтролер фактично міг виконувати лише одну дію, наприклад блимати світлодіодом, і не мав можливості виконувати інші задачі.

Програма «Вітаю, Світе!»

```
org     0x0000
goto    Init

; обробник переривання ---------
org     0x0004
; збереження контексту
movwf   0x0C    ; W_Temp
swapf   STATUS, w
movwf   0x0D    ;STATUS_Temp

; зміна стану порту RB0
movlw   b'00000001'
xorwf   PORTB, f

; очищення прапору переривань
bcf     INTCON, T0IF

; відновлення контексту
swapf   0x0D, w
movwf   STATUS
swapf   0x0C, f
swapf   0x0C, w

; повернення з переривання
retfie
```

```
Init:   ; ініціалізація ---------
    bsf     STATUS, RP0

    ; налаштування порту B
    movlw   b'11111110'
    movwf   TRISB

    ;налаштування таймеру
    movlw   b'11000111'
    movwf   OPTION_REG

    ;налаштування переривань
    movlw   b'10100000'
    movwf   INTCON

    bcf     STATUS, RP0

MainLoop:  ; основний цикл ------
    goto    MainLoop

    END
```

Використання процесорного часу – 0,0214%
(15,25 (разів за секунду) * 14 (машинних циклів) =
= 214 машинних циклів за секунду
(для тактової частоти 4 МГц)

```
        org     0x000

        ; налаштування порту B
        bsf     STATUS, RP0
        movlw   b'11111110'
        movwf   TRISB
        bcf     STATUS, RP0

MainLoop:
        ; засвітити світлодіод
        bsf     PORTB, RB0
        call    Pause

        ; загасити світлодіод
        bcf     PORTB, RB0
        call    Pause

        goto    MainLoop

Pause:  ; затримка (пауза)
        incfsz  0x0C
        goto    Pause
        incfsz  0x0D
        goto    Pause
        return

        END
```

Використання процесорного часу – 100%

Отже, таймери та переривання є базовими механізмами, без яких практично неможливо створювати повноцінні програми для мікроконтролерів. Аналогічні підходи широко використовуються в комп'ютерних системах, де поєднуються основні програми, таймери та механізми переривань. Незалежно від складності задачі, цей підхід дозволяє ефективно використовувати процесорний час і будувати масштабовані рішення. Тому опанування таймерів і переривань є необхідним етапом у вивченні програмування мікроконтролерів.

Висновки

- Таймери та переривання дозволяють звільнити процесорний час на виконання корисних операцій
- Переривання дозволяють перейти до «подійного» методу програмування
- Таймери мікроконтролерів є стандартними периферійними пристроями

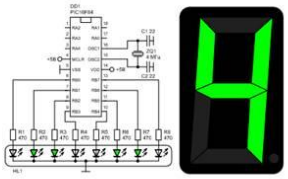
УМОВНІ ОПЕРАЦІЇ

Продовжуємо вивчення дисципліни «Програмування мікроконтролерів та пристроїв інтернету речей». Темою сьогоднішнього заняття є умовні операції. Умовні операції – це команди, або набори команд, які виконуються за певної умови. Тобто, у нас є якась умова, наприклад, чи натиснув користувач певну кнопку, і якщо кнопка натиснена, то мікроконтролер буде виконувати якійсь певні дії, а якщо не натиснена, то дії, які будуть виконуватися мікроконтролером, вже будуть іншими або дій не буде взагалі. Умовні операції дуже часто використовуються як в повсякденному житті, так і в програмному забезпеченні, тому ви просто повинні вміти їх реалізовувати.

Умовні операції ми будемо вивчати, вирішуючи конкретне практичне завдання. На наступній практичній роботі вам потрібно буде сформувати символи на однорозрядному світлодіодному семисегментному індикаторі, підключеному до вже знайомого вам мікроконтролера PIC16F84. Для того, щоб отримати мінімальну позитивну оцінку – 60 балів – вам треба буде послідовно відобразити на індикаторі символи цифр від 0 до 9, а для отримання більш високої оцінки вам доведеться проявити творчий підхід та відобразити більшу кількість символів, або навіть ціле повідомлення. Наприклад, можна відобразити спочатку цифри від 0 до 9, потім латинські літери від A до F. Можна відобразити дату свого народження або дату народження улюбленого песика, або якусь іншу сакральну послідовність символів – тут завдання обмежено лише вашою уявою.

Особливість наступного практичного завдання полягає в тому, що схема підключення індикатора до мікроконтролера залежить від номеру варіанту і вам заздалегідь невідома. Отже, перед тим як почати програмувати вам потрібно буде спочатку експериментально з'ясувати який сегмент індикатора до якого виводу мікроконтролера підключений, а вже потім за допомогою умовних операцій формувати символи. Така побудова макету створює для вас додатковий технічний виклик, а ще запобігає копіюванню програми між студентами, тому що програма, написана для одного варіанту не буде коректно працювати для іншого варіанту схеми.

Завдання на практичну роботу



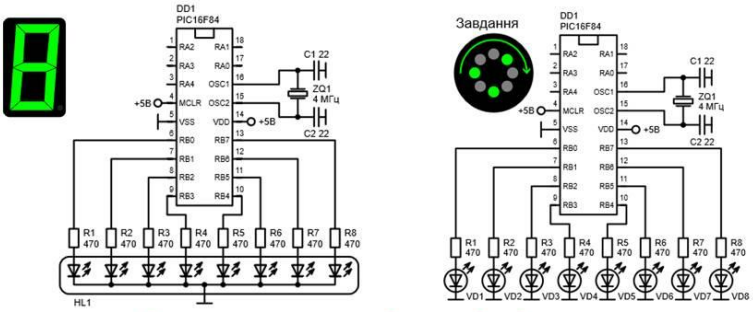
Потрібно написати програму, яка б відображала на семисегментному індикаторі значення змінної, яка циклічно змінюється з 0 до 9

У кожного варіанту своя невідома схема підключення індикатора до мікроконтролера

Електрично наступний лабораторний макет майже повністю ідентичний світловому автомату, з яким ви працювали раніше. Основним елементом схеми є мікроконтролер PIC16F84, що працює на частоті 4 МГц, до порту PORTB якого через струмообмежувальні резистори підключено вісім світлодіодів. Схема підключення світлодіодів така, що для того, щоб їх засвітити, на виводі мікроконтролера потрібно сформувати сигнал з рівнем, що відповідає рівню логічної одиниці. Єдиною різницею є те, що використовуються не окремі світлодіоди з круглими лінзами, як у світловому автоматі, а спеціальний електронний компонент – однорозрядний світлодіодний семисегментний індикатор, всередині якого розташовані сім або вісім світлодіодів, лінзи яких мають спеціальну форму у вигляді сегменту символу. Сім світлодіодів утворюють знакове поле, на якому можна сформувати цифри та деякі символи. А восьмий світлодіод, якого може і не бути, виготовлений у вигляді крапки, розташованій у правому нижньому куті символьного поля. Зазвичай він використовується в багаторозрядних індикаторах для відображення точки, що розділяє цілу та дробову частини числа.

Таким чином, ваше нове завдання дуже схоже на попереднє. Вам потрібно сформувати та періодично змінювати сигнали на виводах мікроконтролера так, щоб користувач побачив на індикаторі символи у певній послідовності. Тому ви можете взяти за основу попередній проєкт, де ви вчилися користуватися таймерами та перериваннями та модифікувати його, відповідно до наступного технічного завдання.

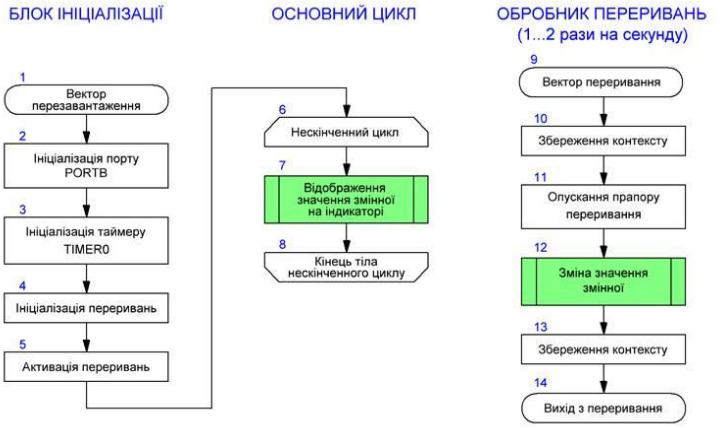
Аналіз електричної схеми



Електрична схема аналогічна схемі цифрового автомату (але який світлодіод якому сегменту індикатора відповідає – невідомо)

Як і в попередньому практичному завданні, я рекомендую вам побудувати програмне забезпечення для мікроконтролера за класичною структурою, яка складається з трьох частин. Перша частина – це блок ініціалізації, який виконується одноразово – одразу після перезавантаження мікроконтролера. У цьому блоці ми готуємо апаратну і програмну частини мікроконтролера до подальшої роботи: налаштовуємо периферійні модулі, ініціалізуємо змінні та виконуємо інші процедури, необхідні для початку роботи.

Рекомендований алгоритм програмного забезпечення

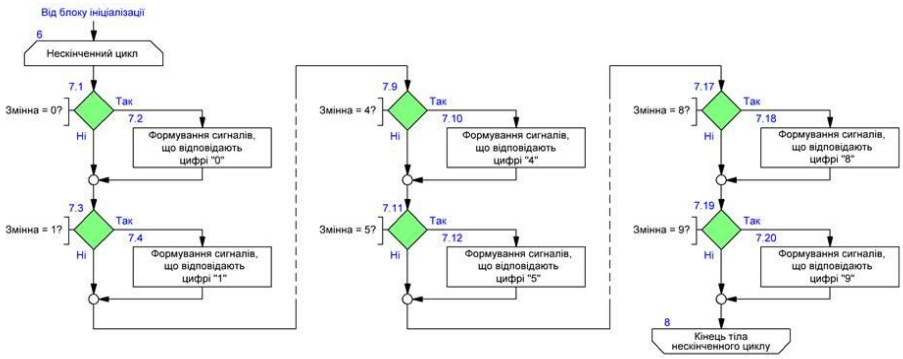


Друга частина програми – це основний нескінченний цикл. У нашому практичному завданні у цій частині програми ми будемо відображати символи на семисегментному індикаторі. Відповідно до теми сьогоднішнього завдання, це буде зроблено за допомогою умовних операцій. Для цього ми створимо спеціальний програмний об’єкт – змінну, яка може приймати значення від 0 до 255, але реальна кількість значень, яку вона буде приймати, буде меншою – наприклад, від 0 до 9.

У нескінченному циклі ми будемо аналізувати значення цієї змінної, і відповідно до нього формувати сигнали на виводах мікроконтролера. Якщо значення змінної дорівнює нулю, то мікроконтролеру потрібно засвітити такі сегменти, щоб користувач побачив цифру «0». Якщо значення змінної дорівнює одиниці, то потрібно засвітити світлодіоди, що сформують символ «1». І так далі – скільки ми хочемо відобразити символів, стільки умов у нас буде в основному циклі програми.

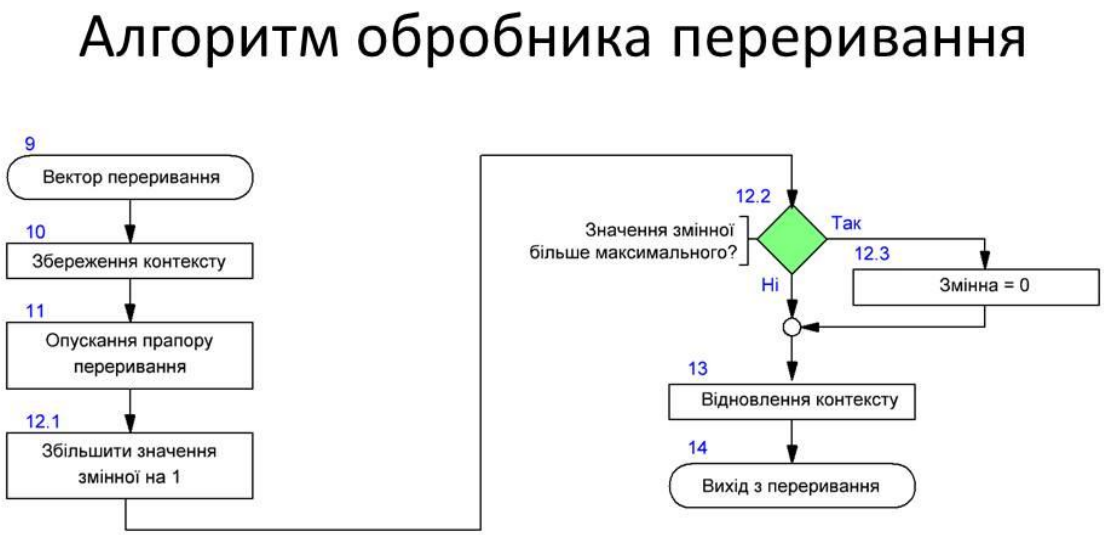
Бачимо, що наш основний цикл складається виключно з умовних операцій. Для отримання мінімальної оцінки нам потрібно реалізувати 10 умовних операцій, а якщо хочете більш високу оцінку, то потрібно буде збільшити і кількість умовних операцій, і кількість комбінацій сигналів, які їм відповідають.

Алгоритм основного циклу



Змінювати значення нашої змінної будемо у третій частині програми – в обробнику переривання від таймера, який ми налаштуємо так, щоб переривання виникало 1…2 рази на секунду. Алгоритм обробника переривання вже вам відомий з попередньої практичної роботи: спочатку потрібно проаналізувати прапори, щоб визначити яке переривання спрацювало, потім опустити прапор переривання, що оброблюється, а вже після цього робити якійсь корисні речі, у нашому випадку – змінювати значення змінної. Я рекомендую наступний алгоритм зміни її значення. Спочатку збільшуємо змінну на одиницю. Потім перевіряємо її значення, і якщо воно більше за максимальне, то встановлюємо змінну у

початкове нульове значення. Отже, якщо ми хочемо відобразити цифри від 0 до 9, то після збільшення змінної на одиницю, нам треба перевірити, чи не стала вона більшою за 9, і якщо це так, то тоді ми встановлюємо цю змінну у значення 0. Отже, і в обробнику переривань у нас також є одна умовна операція.



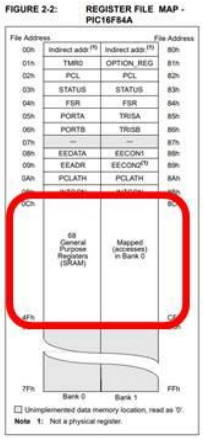
Таким чином, наша програма складається наче з двох окремих процесів. Один процес, що виконується в основному циклі програми, формує сигнали на виводах мікроконтролера так, щоб користувач бачив на індикаторі потрібний символ. Другий процес, що виконується у перериваннях, змінює по колу символи, що відображаються на індикаторі Ці два процеси зв'язані між собою через одну змінну, яка циклічно змінюється раз або двічі на секунду. Для побудови програмного забезпечення використовуємо класичний алгоритм програмного забезпечення для мікроконтролера. Налаштовувати мікроконтролер, працювати з портами введення-виведення, таймерами та перериваннями ми вже вміємо. Залишилося лише два питання, які ви ще поки не знаєте як вирішувати: як створити змінну та як організувати умовні операції.

Змінні та константи

Починаємо з простого питання: як на мовах асемблеру створюються змінні? Спочатку згадаємо, що змінна – це щонайменше одна комірка оперативної пам'яті, призначена для зберігання інформації. Тобто, кожна змінна повинна зв'язуватися з конкретним апаратним об'єктом – щонайменше однією коміркою оперативної пам'яті. Це дуже важливо, тому що кількість комірок оперативної пам'яті обмежена, і відповідно, кількість змінних, якими можна одночасно користуватися в програмі, також обмежена. Тому оперативна пам'ять – це ресурс, який потрібно свідомо розподіляти.

Із попередніх матеріалів ви вже знаєте, що оперативна пам'ять мікроконтролера PIC16F84 поділяється на дві частини. Перша – це регістри спеціального призначення, які апаратно зв'язані з елементами ядра мікроконтролера або периферійними пристроями. Їх адреси знаходяться в діапазонах 0x00…0x0B та 0x80…0x8B. Прикладами регістрів спеціального призначення є STATUS, PORTB, TRISB, OPTION_REC та інші. Зберігати в них змінні або якусь іншу довільну програмну інформацію не можна, бо це одразу негативно вплине на роботу ядра та периферійних модулів мікроконтролера. Друга частина – це регістри загального призначення, адреси яких в знаходяться межах 0x0C…0x4F та 0x8C…0xCF. Ці комірки оперативної пам'яті не пов'язані ні з чим, тому саме вони є фізичним місцем для зберігання змінних.

Визначення змінних



- Змінні повинні знаходитися в регістрах загального призначення (General Purpose Register – GPR)
- Адреси регістрів загального призначення знаходяться в межах 0x0C…0x4F (12…79) – усього 68 комірок (68 байт) пам'яті
- Регістри, що мають адреси (0x8C…0xCF) фізично є комірками, що мають адреси 0x0C…0x4F

Для доступу до регістрів загального призначення не потрібно перемикати біт RP0 у регістрі STATUS

Ви вже знаєте, що в мікроконтролерах PIC середнього сімейства адресний простір оперативної пам'яті поділяється на сторінки розміром 128 байт, які обираються шляхом встановлення відповідного значення бітів RP0 та RP1 у регістрі STATUS. В мікроконтролері PIC16F84 фізично існують тільки дві сторінки оперативної пам'яті, тому при встановленні адресації до уваги приймається лише біт RP0. Значення біту RP1, який активує третю та четверту сторінки оперативної пам'яті, в мікроконтролерах PIC16F84 до уваги не береться.

В мікроконтролері PIC16F84 регістри спеціального призначення розміщені на двох сторінках, тому перед тим як звернутися до комірок, розташованих на другій сторінці, наприклад TRISA або TRISB, біт RP0 у регістрі STATUS повинен дорівнювати одиниці, а якщо потрібно звертатися до комірок, розташованих на першій сторінці, наприклад, PORTA або PORTB, то цей біт повинен дорівнювати нулю.

З комірками загального призначення ситуація така ж сама – вони можуть розташовуватися на різних сторінках оперативної пам'яті, і перед тим, як до них звернутися, потрібно спочатку встановити у правильне значення біти RP0 та RP1 у регістрі STATUS. Але усе не стосується мікроконтролера PIC16F84. У мікроконтролері PIC16F84 є 68 комірок оперативної пам'яті загального призначення, які продубльовано на обох сторінках. Це значить, що кожна з фізичних комірок оперативної пам1яті загального призначення, фактично має дві адреси, і звернення до комірок пам'яті з адресами 0x0C або 0x8C – це звернення до однієї і тої ж фізичної комірки пам'яті. Тому при доступі до комірок оперативної пам'яті загального призначення мікроконтролера PIC16F84 на біти RP0 та RP1 у регістрі STATUS можна не звертати уваги. Але усе це стосується лише мікроконтролера PIC16F84 – в інших моделях правила адресації комірок оперативної пам'яті загального призначення може бути іншим, тому перед тим як розподіляти змінні спочатку уважно вивчіть технічну документацію на конкретну модель мікроконтролера.

Отже, змінна – це щонайменше одна комірка оперативної пам'яті, причому оперативної пам'яті загального призначення – комірки спеціального призначення для зберігання довільної програмної інформації краще не використовувати. Ви вже працювали з комірками оперативної пам'яті загального призначення, тобто ви вже фактично працювали зі змінними, просто про це ще не підозрювали.

Давайте згадаємо саму першу програму, а точніше код, який формував затримку між перемиваннями світлодіода. Тоді ви спочатку збільшували на одиницю комірку пам'яті, що мала адресу 0x0C, а потім – коли відбувалося її переповнення – збільшували на одиницю комірку пам'яті з адресою 0x0D. Таким чином, ви фактично виконували 256 * 256 = 65536 порожніх циклів, які процесор виконував приблизно за 0,2 секунди.

Але давайте подимись уважно на цей код. Ви вже знаєте, що він є повністю працездатним, проте чи буде він зрозумілий іншому програмісту? Та що там інша людина, ви самі через місяць навряд чи будете не будете пам'ятати, що знаходиться в комірці за адресою 0x0C, а що – в комірці за адресою 0x0D, тому що адреси – це просто цифри, які навіть для досвідченого програміста мало що значать.

Робота з регістрами загального призначення

Поганий приклад

```
Loop1
        incfsz  0x0C, f
        goto    Loop1
        incfsz  0x0D, f
        goto    Loop1
```

Гарний приклад

```
Loop1
        incfsz  CntLow, f
        goto    Loop1
        incfsz  CntHigh, f
        goto    Loop1
```

Як назначити ім'я регістру?

А чи можемо ми дати смислові імена коміркам пам'яті. Звісно можемо! Нехай комірка пам'яті, що має адресу 0x0C буде називатися CntLow, а комірка, що має адресу 0x0D – CntHigh. Ці назви є скороченнями від англійський словосполучень, відповідно, Counter Low Byte і Counter High Byte, тобто молодший та старший байт лічильника. Таким чином, навіть у першій програмі у нас вже могла бути достатньо складна 16-розрядна змінна Counter, яка займала би дві суміжні комірки оперативної пам'яті з адресами 0x0C та 0x0D, причому молодший байт цієї змінної мав окрему назву CntLow і розташовувався за адресою 0x0C, а старший байт мав назву CntHigh та розташовувався за адресою 0x0D.

Тепер ми бачимо, що наш код виглядає зовсім інакше. Замість незрозумілих адрес фізичної пам'яті ми показуємо, в першу чергу – самому собі, що у нас є якийсь лічильник, який щось там десь лічить чи рахує. Тому якщо надати коміркам оперативної пам'яті смислові назви, то ми зможемо істотно зменшити кількість помилок при написанні програмного коду, та скоротити час на написанні програмного забезпечення. Тепер розберемося як це можна зробити.

Є кілька способів надання синонімів у програмному забезпеченні, що пишеться на мовах асемблеру. Потрібно лише пам'ятати, що синоніми потрібно описувати до першого їх використання у коді, інакше компілятор скаже, що у цьому рядку є помилка. Хорошим варіантом є створення переліку таких назв в окремому файлі. У вас вже є такий файл, який називається P16F84.INC – ви його використовуєте з самого першого проєкту. У цьому файлі розробники IDE MCLAB надали назви усім регістрам спеціального призначення. Більше того, у цьому файлі надано назви навіть окремим бітам регістрів спеціального призначення. Такий підхід не тільки прискорює створення написання програмного коду, а ще й значно пришвидшує перенос коду на інший мікроконтролер. Наприклад, у мікроконтролері PIC16F84 регістр спеціального призначення STATUS має адресу 0x03, а в іншій моделі мікроконтролера – цей регістр буде мати аналогічний функціонал, але розташовуватись за іншою адресою, наприклад, 0x04. У цьому випадку в коді достатньо один раз в описанні синоніму виправити адресу не чіпаючи основного коду програми.

Є три найбільш поширені варіанти надання назв коміркам оперативної пам'яті: за допомогою директиви EQ (скорочення від «Equal» – дорівнює), можна використовувати директиву DEFINE (визначити), а можна використовувати конструкцію CONSTANT. Синтаксис цих директив схожий. Між цими визначеннями є різниці, але вони виходять за межі сьогоднішнього заняття. Тому, щоб не витрачати час, просто прийміть, що змінні можна визначати будь-яким із цих способів. Але я вам рекомендую при програмуванні на мовах асемблеру використовувати четвертий спосіб.

Варіанти призначення імен змінним

```
#include <p16F84A.inc>

__CONFIG _XT_OSC & _WDT_OFF

CntLow  equ 0x0C
CntHigh equ 0x0D

org     0x000

bsf     STATUS, RP0
movlw   b'00000000'
movwf   TRISB
bcf     STATUS, RP0
movwf   PORTB

MainLoop

bsf     PORTB, 0

Loop1
incfsz  CntLow, f
goto    Loop1
incfsz  CntHigh, f
goto    Loop1

bcf     PORTB, 0
```

```
#include <p16F84A.inc>

__CONFIG _XT_OSC & _WDT_OFF

#define CntLow  0x0C
#define CntHigh 0x0D

org     0x000

bsf     STATUS, RP0
movlw   b'00000000'
movwf   TRISB
bcf     STATUS, RP0
movwf   PORTB

MainLoop

bsf     PORTB, 0

Loop1
incfsz  CntLow, f
goto    Loop1
incfsz  CntHigh, f
goto    Loop1

bcf     PORTB, 0
```

```
#include <p16F84A.inc>

__CONFIG _XT_OSC & _WDT_OFF

constant CntLow = 0x0C
constant CntHigh = 0x0D

org     0x000

bsf     STATUS, RP0
movlw   b'00000000'
movwf   TRISB
bcf     STATUS, RP0
movwf   PORTB

MainLoop

bsf     PORTB, 0

Loop1
incfsz  CntLow, f
goto    Loop1
incfsz  CntHigh, f
goto    Loop1

bcf     PORTB, 0
```

Четвертий спосіб полягає у використанні клнструкції CBLOCK (Constant Block – блок констант). Коли ви використовуєте CBLOCK для визначення змінних, то ви вказуєте лише початкову адресу розташування регістрів спеціального призначення. Далі просто перераховуєте назви змінних, яким компілятор вже сам обчислить адреси. Тобто, у коді, що наведено на екрані, змінній CntLow компілятор сам призначить адресу 0x0C, змінній CntHigh – адресу 0x0D тощо. При цьому ви мінімально втручаєтеся у цей процес – за вас це робить компілятор, і це додатково зменшує кількість імовірних помилок. Річ у тому, що при наданні кожній змінній адреси вручну можна легко помилитися, особливо якщо у вас багато змінних. Уявіть, що ви використовуєте усі 68 комірок оперативної пам’яті – якщо ви заплутаєтесь і кільком змінним призначите одну і ту саму адресу, то будете довго шукати помилку. Тому краще вказати початкову адресу, а далі розташовувати змінні, як хочете. Потім можна замінити початкову адресу і всі змінні автоматично змінять свої адреси. Отже, рекомендую використовувати конструкцію CBLOCK.

Корисні поради

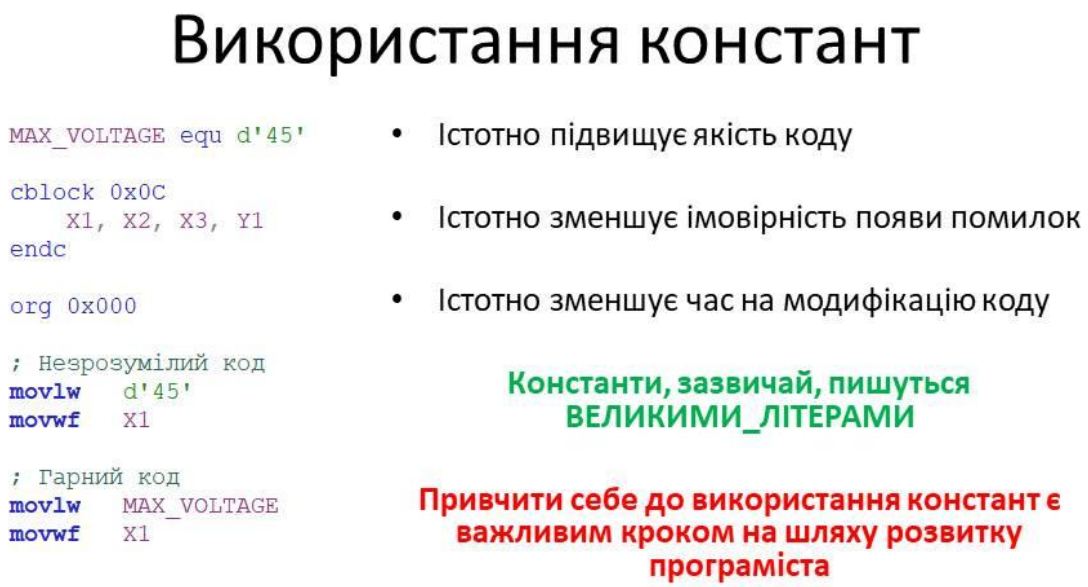
```
#include <p16F84A.inc>

CONFIG _XT_OSC & _WDT_OFF

cblock 0x0C
    CntLow
    CntHigh
endc

org     0x000

bsf     STATUS, RP0
movlw   b'00000000'
movwf   TRISB
bcf     STATUS, RP0
movwf   PORTB

MainLoop

bsf     PORTB, 0

Loop1
incfsz  CntLow, f
goto    Loop1
incfsz  CntHigh, f
goto    Loop1

bcf     PORTB, 0
```

- Змінні визначаються в одному місці – на самому початку файлу вихідного коду або в окремому файлі
- Для визначення імен краще використовувати конструкцію **cblock**
- Імена повинні мати певне смислове навантаження (cnt, temp, voltage, current тощо)

На асемблері немає типів змінних
(усі регістри є беззнаковими цілими числами)

Тепер поговоримо про константи. Без констант можна обійтися, але для кращого розуміння того, що відбувається в програмі, краще використовувати константи. Для констант можна використовувати ті ж конструкції – DEFINE, EQ, CONSTANT. Але на відміну змінних, які використовують апаратні ресурси мікроконтролера, константи є частиною програмного коду, і їх краще визначати вручну за допомогою конструкцій DEFINE, EQ, CONSTANT. Конструкцію CBLOCK для визначення констант використовувати немає сенсу.

Давайте розглянемо конкретний приклад, що демонструє усю потужність використання констант. В коді, який зображено на екрані, змінній X1 присвоюється значення 45. Дивлячись на цей код, незрозуміло, що це за число. Чому 45, а не 46 або 49, і на що впливає це значення? Коли стороння людина дивиться на код, вона не розуміє, та й ви самі через місяць не будете пам'ятати, що означає 45. Але якщо ви призначите цьому значенню конкретний сенс, наприклад, максимальна напруга, то код стає набагато зрозумілішим – тепер ми бачимо, що змінній X1 присвоюється значення якоїсь максимальної напруги. Ні мінімальної, ні середньої – я саме максимальної. Ні струму, ні частоти – а саме напруги. Крім того, для одного пристрою максимальна напруга може дорівнювати 45, а для іншого – 145. Якщо цей параметр в коді використовується більше ніж один раз, то вам – при переносі коду – доведеться довго шукати місця, де є цифра 45, при чому цифра 45, яка відповідає саме максимальній напрузі, а не якомусь іншому параметру, і замінювати її на 145.



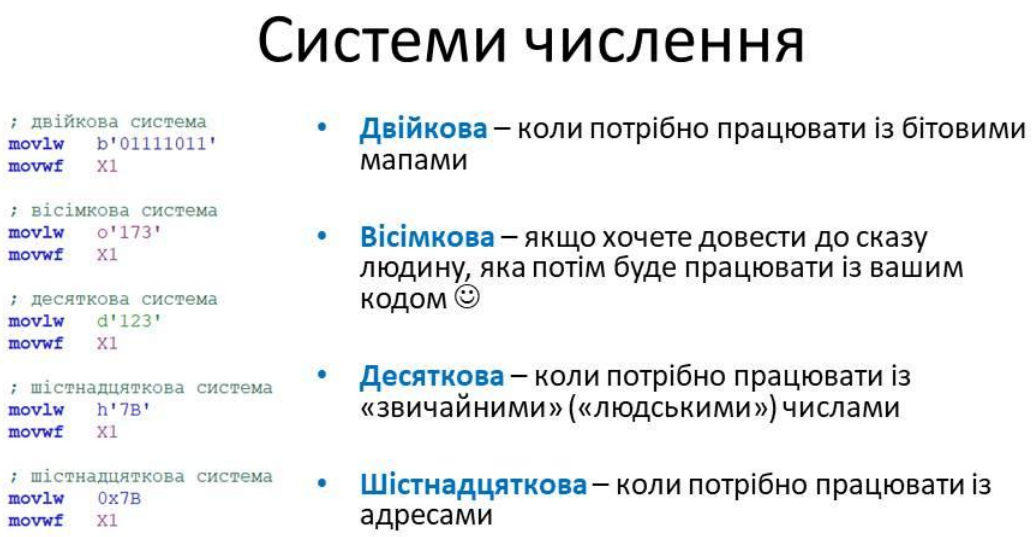
Тому краще завжди використовувати константи. Привчіть себе до цього – це важливо, щоб стати гарним програмістом. Зазвичай константи пишуться великими літерами, це не обов'язково, але коли ви бачите в коді щось написане великими літерами, ви розумієте, що це константа, незмінна величина, яка визначена в одному місці програми.

Хто програмував на мовах високого рівня, той знає, що константи можуть бути різних типів. Проте, якщо мова заходить про написання програм на мовах асемблеру, то в цьому програмному коді частіше за все використовуються числові константи. А при запису числових значень головне правильно обрати систему числення. При написанні програм для мікроконтролерів найбільш поширеними є двійкова, десяткова та шістнадцяткова системи числення.

Десяткова система числення знайома нам з дитинства. Це наша звична система числення, в якій ми виконуємо усі математичні розрахунки. Тому коли нам потрібно вказувати якійсь параметри, що ототожнюються з людським сприйняттям, то десяткова система числення для цього підходить якнайкраще. Ми розглядали приклад, коли для визначення максимальної напруги була використана саме десяткова система числення – вона є найбільш зручною для людини.

Найбільш зручною системою числення для побудови електронних комп'ютерів є двійкова. Створення арифметико-логічних пристроїв, що працюють в двійковій системі числення, технічно є найпростішим. Якщо на якомусь електричному проводі є електрична напруга, то ми вважаємо, що на цій лінії присутній сигнал с рівнем логічної одиниці, а якщо напруги немає – то рівень логічного нуля. Визначати якійсь поміжні рівні напруги, щоб реалізувати більш складні системи числення, виявилося технічно складно, тому усі сучасні електричні комп'ютери працюють з двійковою системою числення.

При програмування мікроконтролерів двійкова система числення використовується для записів чисел, які називають бітовими мапами. Кожен розряд двійкового числа може запам'ятати лише один біт інформації – це дуже мало. Тому усі мікроконтролери мають розрядність більше одиниці, що дозволяє за один інтервал часу оброблювати більшу кількість інформації. Існують 4-, 8-, 16-, 32- та 64-розрядні мікроконтролери і навіть контролери з більшою розрядністю. Наприклад, мікроконтролер PIC16F84 є 8-розрядним – у нього кожен апаратний вузол, пов'язаний із обробкою інформації, має вісім однакових ліній, кожна з яких може передавати 1 біт інформації.



Проте, коли почали створювати електронні комп'ютери, то швидко з'ясувалося, що

Кожна 8-розрядна комірка пам'яті може мати $2^8 = 256$ можливих комбінацій значень – від 0 до 255. Проте в деяких випадках – такого значення є забагато. У цьому випадку інформацію зберігають в 8-розрядних комірках у вигляді бітових мап – коли кожен біт має певне функціональне значення. Ми вже стикалися з бітовими мапами, коли створювали наші програми. Класичними бітовими мапами є регістри TRISA, TRISB, PORTA та PORTB. У цих регістрах кожен біт відповідає за стан певної лінії – виводу мікроконтролера. Тому коли ми працюємо в комірками, де інформація зберігається у вигляді бітових мап, то краще використовувати константи, написані у двійковій системі числення – це дозволяє наочно побачити стан кожного біту.

Однак двійкова система числення вкрай незручна для записів великих чисел. За розрядності більшої за 8 при написанні чисел у двійковому форматі можна легко заплутатися в нулях та одиницях, томі для компактного запису чисел, що використовуються в комп'ютерній техніці використовують шістнадцяткову систему числення. У шістнадцятковій системі числення кожен символ може приймати 16 значень: десять цифр – від 0 до 9 та шість латинських літер – A – F. При використанні шістнадцяткової системи довжина двійкових чисел скорочується в 4 рази – кожні 4 цифри двійкового числа замінюються одним символом шістнадцяткової. У шістнадцятковій системі зручно записувати суто «комп'ютерні» числа, що є похідними від двійкової, наприклад, адреси комірок оперативної пам'яті чи пам'яті програм. Тому шістнадцяткова система є більш поширенішою ніж двійкова. Наприклад, деякі платформи для розробки програмного забезпечення на мовах високого рівня не підтримують двійкову систему, проте усі без винятку підтримують десяткову та шістнадцяткову системи.

І наостанок слід згадати вісімкову систему. У цій системі у кожному розряді числа можуть використовуватися тільки цифри 0 – 7. При цьому кожен розряд числа, записаного у вісімковій системі, відповідає трьом розрядам числа, записаного у двійковій. Вісімкову

систему колись також пробували використовувати при створенні комп'ютерів, але зараз її повністю витіснила більш зручна шістнадцяткова система. На сьогодні використання вісімкової системи обґрунтоване лише тоді, коли ви хочете ускладнити життя людині, яка буде розбиратися в вашому вихідному коді.

В середовищі MPLAB IDE при записі запису числа у вихідному коді програми перед самим числом треба вказати літеру, що позначає систему числення: для двійкової це літера B (binary), для вісімкової – O (octal), для десяткової – D (decimal), а для шістнадцядкової – H (hexadecimal) або префікс «0x». Редактор коду навіть підсвічує ці константи різними кольорами, щоб вам було зручніше його писати та розуміти.

А тепер головне. У середовищі MPLAB IDE системою числення за замовчуванням є шістнадцяткова. Тобто якщо ви не визначите явно систему числення, то компілятор це число буде розглядати як число, записане в шістнадцятковій системі. Тому, якщо ви напишете у коді число 23, то компілятор його інтерпретує як число «35» в десятковій системі. Це дуже підступна помилка яку не завжди можна швидко знайти. Наприклад, число 10 – це насправді число 16 в десятковій системі. Тому при написанні чисел привчіть себе завжди явно вказувати систему числення.

І наостанок про роботу з багатобайтними числами. Мікроконтролер PIC16F84 є 8-розрядним, тобто його арифметико-логічна частина може працювати лише з числами від 0 до 255. Це дуже малий діапазон. Для чисел більше за 255 використовуються багатобайтні числа. Тому при використання багатобайтних чисел і констант на 8-розрядних платформах часто є необхідність звернення до окремих байт цих чисел. При використання констант для цього використовуються директиви Low, High та Upper. Директива Low повертає молодший (нульовий) байт багатобайтної константи, директива High – перший (після нульового) байт, а Upper – повертає другий (після нульового) байт багатобайтної константи. Отже, якщо у нас є якась багатобайтна константа, що має більше ніж 8 розрядів, наприклад 0xFFB8FA56, то директива Low(0xFFB8FA56) поверне 0x56, High(0xFFB8FA56) = 0xFA, а Upper(0xFFB8FA56) = 0xB8.

Використання директив Low, High, Upper

```
MAX_ADDRESS equ 0xFFB8FA56

cblock 0x0C
    X1, X2, X3, Y1
endc

org 0x000

movlw   Low(MAX_ADDRESS)
movwf   X1

movlw   High(MAX_ADDRESS)
movwf   X2

movlw   Upper(MAX_ADDRESS)
movwf   X3

; X1 = 0x56
; X2 = 0xFA
; X3 = 0xB8
```

- **Low**(константа) – повертає нульовий (молодший) байт багатобайтної константи
- **High**(константа) – повертає перший байт багатобайтної константи
- **Upper**(константа) – повертає другий байт багатобайтної константи

Отже, підведемо підсумки. У нашій програмі нам потрібна одна змінна та одна константа. Нехай наша змінна буде розташовуватися в першій комірці оперативної пам'яті загального призначення, що має адресу 0x0C. Для визначення змінної краще використовувати конструкцію CBLOCK, проте це не обов'язково – тут можна використовувати конструкції DEFINE, EQ або CONSTANT. Ім'я змінної може бути любим, наприклад «A», проте не лінуйтесь давати змінним довгі осмислені назви – це потім значно скоротить час на неминучу модифікацію програми.

У якості константи нам треба визначити максимальне значення нашої змінної. Для отримання мінімальної оцінки вам потрібно показати 10 символів, отже ваша змінна буде приймати значення від 0 до 9, тому значення вашої константи буде дорівнювати 9. Константу можна сформувати будь якою з директив DEFINE, EQ або CONSTANT – це діло смаку програміста, який пише код – на розмір коду та на швидкість його виконання це не вплине жодним чином.

Рекомендований код для змінних та констант

```
constant    MAX_NUMBER = d'9'

cblock  0x0C
    Number
endc
```

Умовні операції вже використовувалися раніше. Під час реалізації затримок у першій програмі виконувалося збільшення або зменшення на одиницю комірки оперативної пам'яті з подальшим пропуском наступної команди, якщо результат дорівнював нулю. В асемблері використовується той самий підхід: на апаратному рівні перевіряється умова, і залежно від того, виконується вона чи ні, наступна асемблерна команда або виконується, або пропускається.

У мові асемблеру мікроконтролерів PIC існує чотири асемблерні команди, що реалізують умовні операції. Дві з них вже відомі – **incfsz** та **decfsz**. Ці команди збільшують або зменшують на одиницю вказану комірку пам'яті, і якщо результат цієї операції дорівнює нулю, наступна команда пропускається.

Команди incfsz, decfsz

DECFSZ	Decrement f, Skip if 0
Syntax:	[*label*] DECFSZ f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f) - 1 → (destination); skip if result = 0
Status Affected:	None
Description:	The contents of register 'f' are decremented. If 'd' is 0, the result is placed in the W register. If 'd' is 1, the result is placed back in register 'f'. If the result is 1, the next instruction is executed. If the result is 0, then a NOP is executed instead, making it a 2TCY instruction.

INCFSZ	Increment f, Skip if 0
Syntax:	[*label*] INCFSZ f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f) + 1 → (destination), skip if result = 0
Status Affected:	None
Description:	The contents of register 'f' are incremented. If 'd' is 0, the result is placed in the W register. If 'd' is 1, the result is placed back in register 'f'. If the result is 1, the next instruction is executed. If the result is 0, a NOP is executed instead, making it a 2TCY instruction.

За допомогою команд інкременту та декременту можна реалізувати різні умовні операції. Розглядається еквівалентна логіка, подібна до коду на С, на рівні загального розуміння.

Під час інкременту значення збільшується на одиницю, а результат зберігається у вказане місце. У команд інкременту є додатковий операнд, який визначає, куди зберігати результат: у комірку пам’яті або в акумулятор, який є окремим регістром. Якщо результат зберігається в ту саму комірку пам’яті, то змінна безпосередньо модифікується, що еквівалентно операції збільшення змінної на одиницю. Після цього на апаратному рівні виконується перевірка: якщо результат дорівнює нулю, наступна команда переходу пропускається і виконується код після неї, якщо результат не дорівнює нулю, команда переходу виконується і відбувається перехід на відповідну мітку.

У другому варіанті результат інкременту зберігається в акумуляторі. У цьому випадку сама змінна не змінюється. Нульовий результат після такого інкременту можливий лише тоді, коли початкове значення змінної дорівнює 255, оскільки після збільшення на одиницю відбувається переповнення і результат стає нульовим. У цьому випадку команда переходу пропускається і виконується відповідний код, а якщо значення не дорівнює 255, виконується перехід на задану мітку.

Команда декременту працює аналогічно, але значення зменшується на одиницю. Якщо результат дорівнює нулю, виконується відповідний код. При збереженні результату в комірку пам’яті змінна модифікується, а при збереженні в акумуляторі змінна не змінюється. Різниця лише в одному операнді суттєво змінює поведінку програми, тому під час роботи з асемблером необхідна максимальна уважність і концентрація.

Приклади використання команд incfsz, decfsz

```
A++;
if (A == 0)
{
  // якийсь код
}
```

```
if (A == 255)
{
  // якийсь код
}
```

```
A--;
if (A == 0)
{
  // якийсь код
}
```

```
if (A == 1)
{
  // якийсь код
}
```

```
    incfsz  A, f
    goto    EndIf

    ; якийсь код

EndIf:
    ; подальший код
```

```
    incfsz  A, w
    goto    EndIf

    ; якийсь код

EndIf:
    ; подальший код
```

```
    decfsz  A, f
    goto    EndIf

    ; якийсь код

EndIf:
    ; подальший код
```

```
    decfsz  A, w
    goto    EndIf

    ; якийсь код

EndIf:
    ; подальший код
```

Змінюється стан акумулятора (w)

Змінюється стан акумулятора (w)

Є ще дві аналогічні команди, які перевіряють стан біту в комірці оперативної пам’яті. Це BTFS і BTFSC. Вони працюють за тим самим принципом: вказується адреса оперативної пам’яті, це може бути комірка спеціального або загального призначення, після чого перевіряється заданий біт. В одному випадку, якщо біт дорівнює одиниці, пропускається наступна команда. В іншому випадку, якщо біт дорівнює нулю, також пропускається наступна команда.

Таким чином умовні операції в асемблері PIC реалізуються лише цими чотирма командами, інших можливостей немає. Питання полягає в тому, які саме біти перевіряти, якщо в комірці пам’яті зберігається число, наприклад 0, 1, 2, 3, 4.

Команди btfss, btfsc

BTFSS	Bit Test f, Skip if Set
Syntax:	[*label*] BTFSS f,b
Operands:	$0 \le f \le 127$ $0 \le b < 7$
Operation:	skip if (f<b>) = 1
Status Affected:	None
Description:	If bit 'b' in register 'f' is '0', the next instruction is executed. If bit 'b' is '1', then the next instruction is discarded and a NOP is executed instead, making this a 2TCY instruction.

BTFSC	Bit Test, Skip if Clear
Syntax:	[*label*] BTFSC f,b
Operands:	$0 \le f \le 127$ $0 \le b \le 7$
Operation:	skip if (f<b>) = 0
Status Affected:	None
Description:	If bit 'b' in register 'f' is '1', the next instruction is executed. If bit 'b' in register 'f' is '0', the next instruction is discarded, and a NOP is executed instead, making this a 2TCY instruction.

Які біти перевіряти, якщо у нас ЧИСЛО!!!

Команди BTFSS і BTFSC є універсальними для реалізації умовних перевірок. Якщо потрібна довільна умова, використовуються саме ці команди. Інкремент і декремент є більш специфічними і застосовуються лише в окремих ситуаціях, коли це доречно, оскільки код у такому випадку виходить компактнішим і швидше виконується. В загальному випадку для умовних операцій застосовуються саме команди перевірки бітів.

Перевірка виконується над бітами, розташованими в регістрі статусу. Ці біти є прапорами арифметико-логічних операцій. Їх три. Прапор Z встановлюється апаратно, якщо результат арифметико-логічної операції дорівнює нулю. Прапор C використовується тоді, коли в результаті арифметико-логічної операції виникає переповнення регістру. Прапор DC відповідає переповненню при арифметико-логічних операціях над тетрадами.

Прапор DC є спадком від чотирирозрядних мікроконтролерів і використовується для збереження сумісності зі старими системами. Він актуальний лише при роботі з чотирирозрядними числами і в сучасних восьмирозрядних мікроконтролерах практично не використовується. На практиці для умовних операцій зазвичай перевіряються прапори Z і C, тобто результат, що дорівнює нулю, або факт виникнення переповнення.

Розглядаються приклади реалізації умов. Робота ведеться лише з цілими беззнаковими числами. Можливі чотири перевірки: дорівнює, не дорівнює, більше, менше.

Для виконання перевірки два числа спочатку піддаються арифметико-логічній операції. Це може бути додавання, віднімання або, найчастіше, віднімання чи операція виключного АБО. У результаті формується значення, яке саме по собі не використовується і зазвичай зберігається в акумуляторі, після чого перезаписується. Важливими є не дані результату, а прапори, які встановлюються після виконання операції. Саме за станом цих прапорів аналізується співвідношення між двома числами. Числа можуть бути константами з пам’яті програм або значеннями змінних, але операції в будь-якому випадку виконуються через акумулятор.

Регістр STATUS

STATUS REGISTER (ADDRESS 03h, 83h)

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

bit 2 **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC**: Digit carry/$\overline{\text{borrow}}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions) (for $\overline{\text{borrow}}$, the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0 **C**: Carry/$\overline{\text{borrow}}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions) (for $\overline{\text{borrow}}$, the polarity is reversed)
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

Для перевірки на рівність значення з комірки A завантажується в акумулятор, після чого від нього віднімається константа або інше число. Якщо ці значення рівні, результат віднімання дорівнює нулю і встановлюється прапор Z. При встановленому прапорі Z команда переходу пропускається і виконується відповідний код. Якщо числа не рівні, прапор Z скинутий і виконується перехід на іншу мітку.

Аналогічно реалізується перевірка на нерівність, але змінюється лише умова перевірки прапора Z. В одному випадку пропуск команди відбувається при встановленому прапорі Z, в іншому – при скинутому. Таким чином код виконується тоді, коли два числа відрізняються одне від одного. Для перевірок більше або менше використовується прапор C.

Приклади умовних операцій

```
if (A == ЧИСЛО)
{
  // якийсь код
}
```

```
    movfw   A
    sublw   'ЧИСЛО'
    btfss   STATUS, Z
    goto    EndIf

    ; якийсь код

EndIf:
    ; подальший код
```

```
if (A != ЧИСЛО)
{
  // якийсь код
}
```

```
    movfw   A
    sublw   'ЧИСЛО'
    btfsc   STATUS, Z
    goto    EndIf

    ; якийсь код

EndIf:
    ; подальший код
```

```
if (A > ЧИСЛО)
{
  // якийсь код
}
```

```
    movfw   A
    sublw   'ЧИСЛО'
    btfsc   STATUS, C
    goto    EndIf

    ; якийсь код

EndIf:
    ; подальший код
```

```
if (A <= 1)
{
  // якийсь код
}
```

```
    movfw   A
    sublw   'ЧИСЛО'
    btfss   STATUS, C
    goto    EndIf

    ; якийсь код

EndIf:
    ; подальший код
```

Виконується арифметична операція віднімання W = ЧИСЛО – A
Якщо A менше або дорівнює ЧИСЛУ, тоді C = 1 (W ≥ 0), інакше C = 0 (W < 0)
Якщо A дорівнює ЧИСЛУ, тоді Z = 1 (W = 0), інакше Z = 0 (W ≠ 0)

Розглядається робота команди віднімання. Хоча мікроконтролер є восьмирозрядним, під час виконання арифметико-логічних операцій перший операнд фактично обробляється як дев'ятирозрядний.

Перед виконанням операції віднімання мікроконтролер апаратно встановлює прапор C, і цей прапор використовується як дев'ятий розряд числа, від якого виконується віднімання. Тобто віднімання виконується не від восьмирозрядного значення, а від дев'ятирозрядного.

Наприклад, в акумуляторі знаходиться число з комірки пам'яті, яке порівнюється з константою. Якщо в акумуляторі значення 15 і від нього віднімається 5, фактично віднімання виконується від значення 271. Результатом буде 266, але в акумуляторі зберігається лише молодші вісім біт, тобто значення 10. Прапор C при цьому залишається встановленим.

Якщо обидва числа рівні, результат в акумуляторі дорівнює нулю, хоча з урахуванням дев'ятого біта фактичний результат дорівнює 256. Прапор C у цьому випадку також встановлений.

Якщо ж віднімається число, більше за значення в акумуляторі, виникає займання. Фактично віднімання виконується від 271, але результат супроводжується скиданням прапора C. Таким чином, якщо віднімане число більше за значення в акумуляторі, прапор C скидається, а якщо менше або дорівнює – прапор C залишається встановленим.

Механізм роботи віднімання потребує уважного розуміння, але сам алгоритм є простим. Ключовим є правильний вибір прапора для перевірки умови. Для перевірок на рівність або нерівність у будь-якому випадку використовується прапор Z. Для перевірок більше або менше використовується прапор C.

Якщо в умові була використана неправильна команда або перевірено не той прапор, це легко виявити під час налагодження в симуляторі і швидко виправити. Попри це, принцип роботи бажано зрозуміти хоча б один раз, оскільки в окремих ситуаціях це необхідно для коректної реалізації умовних операцій. Таким чином, використовуючи перевірку прапорів Z і C, можна реалізувати будь-яку умовну операцію.

Під час реалізації умовних операцій важливо правильно обрати арифметико-логічну операцію. Необхідно розуміти, які саме команди використовуються і як вони впливають на стан прапорів.

У технічній документації показано, що найчастіше використовується прапор Z, тобто ознака нульового результату. Також у практиці застосовується прапор C, стан якого використовується для аналізу результатів арифметико-логічних операцій.

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	fff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00	0101	dfff	fff	Z	1,2
CLRF	f	Clear f	1	00	0001	1fff	fff	Z	2
CLRW	-	Clear W	1	00	0001	0xxx	xxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	fff	Z	1,2
DECF	f, d	Decrement f	1	00	0011	dfff	fff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1 (2)	00	1011	dfff	fff		1,2,3
INCF	f, d	Increment f	1	00	1010	dfff	fff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1 (2)	00	1111	dfff	fff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	fff	Z	1,2
MOVF	f, d	Move f	1	00	1000	dfff	fff	Z	1,2
MOVWF	f	Move W to f	1	00	0000	1fff	fff		
NOP	-	No Operation	1	00	0000	0xx0	000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	fff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	fff	C	1,2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	fff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	fff		1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	fff	Z	1,2

Для перевірок на рівність або нерівність простіше використовувати побітову операцію виключного АБО. Існують дві відповідні команди: порівняння акумулятора з константою та порівняння акумулятора з коміркою пам’яті. Ці команди впливають лише на прапор Z.

Побітове виключне АБО

Виключне АБО

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Y = X1 ^ X2

XORLW	Exclusive OR Literal with W
Syntax:	[*label*] XORLW k
Operands:	$0 \le k \le 255$
Operation:	(W) .XOR. k → (W)
Status Affected:	Z
Description:	The contents of the W register are XOR'ed with the eight-bit literal 'k'. The result is placed in the W register.

XORWF	Exclusive OR W with f
Syntax:	[*label*] XORWF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(W) .XOR. (f) → (destination)
Status Affected:	Z
Description:	Exclusive OR the contents of the W register with register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in register 'f'.

Побітове виключне АБО використовується у випадках, коли треба перевірити два числа на рівність (A = B або A ≠ B)

Таким чином формується проста схема перевірки. Якщо два числа рівні, результат операції виключного АБО дорівнює нулю і прапор Z встановлюється. Якщо числа не рівні, результат буде ненульовим і прапор Z буде скинутий.

Команда виключного АБО простіша для сприйняття і використання, ніж команда віднімання. Тому в більшості практичних випадків, зокрема в циклах, де перевіряється рівність значення одиниці або інша умова дорівнює чи не дорівнює, доцільно застосовувати виключне АБО. Для умов більше або менше необхідно використовувати віднімання.

Побітове Виключне АБО

Біт Z: 0

0	1	0	0	1	1	0	1	Операнд 1
0	0	1	0	0	1	1	1	Операнд 2
0	1	1	0	1	0	1	0	Результат

Біт Z: 1

0	1	0	0	1	1	0	1	Операнд 1
0	1	0	0	1	1	0	1	Операнд 2
0	0	0	0	0	0	0	0	Результат

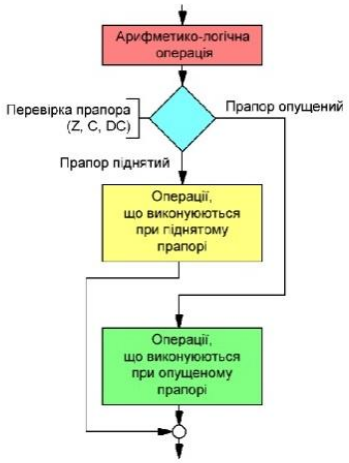
- Використовується для того перевірити значення регістру або акумулятора на певне число
- Прапор Z буде встановлений коли обидва операнди мають однакові значення (тоді результатом ииключного АБО буде нуль)

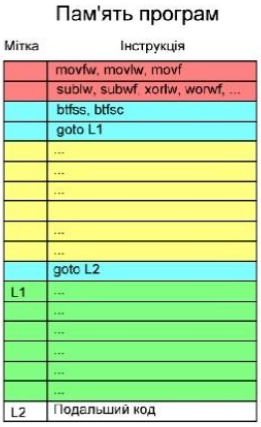
Класична умовна операція з двома гілками має стандартну структуру. Спочатку виконується арифметико-логічна операція, зазвичай за допомогою двох асемблерних команд. Після цього перевіряється стан одного з прапорів за допомогою команд BTFSS або BTFSC. Далі розміщується команда безумовного переходу на одну з гілок.

Якщо команда переходу не пропускається, виконується блок команд для однієї гілки, після чого обов’язково виконується безумовний перехід на кінець обробника. Якщо ж умова не виконується і команда переходу пропускається, виконується інший блок команд, після чого виконання автоматично переходить до подальшого коду. Таким чином умовна операція з двома гілками реалізується в пам’яті програм за допомогою асемблерних команд.

Класична умовна операція

```
    ; перевірка
    ; Number = 0?
    movfw   Number
    xorlw   d'0'
    btfss   STATUS, Z
    goto    L1

    ; прапор Z піднятий
    ; Number = 0
    movlw   b'00111111'
    movwf   PORTB
    goto    L2

    ; прапор Z опущений
    ; Number <> 0
L1: movlw   b'00000110'
    movwf   PORTB

    ;подальший код
L2: movfw   Number
    xorlw   d'1'
```

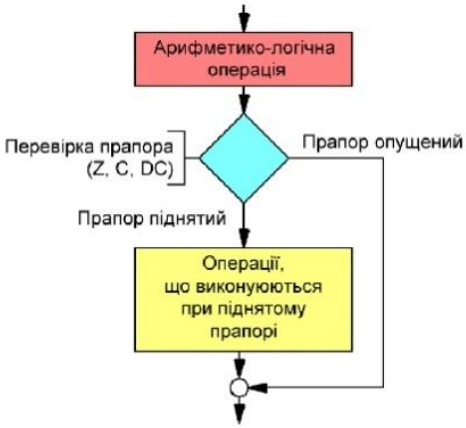




Якщо умовна операція має лише одну гілку, структуру можна спростити і зекономити команди. У цьому випадку при невиконанні умови одразу виконується перехід на подальший код. Такий підхід є прямим спрощенням класичної умовної операції і не містить додаткових складнощів.

Операція з однією умовною гілкою

```
    ; перевірка
    ; Number = 0?
    movfw   Number
    xorlw   d'0'
    btfss   STATUS, Z
    goto    L1

    ; прапор Z піднятий
    ; Number = 0
    movlw   b'00111111'
    movwf   PORTB

    ;подальший код
L1: movfw   Number
    xorlw   d'1'
```

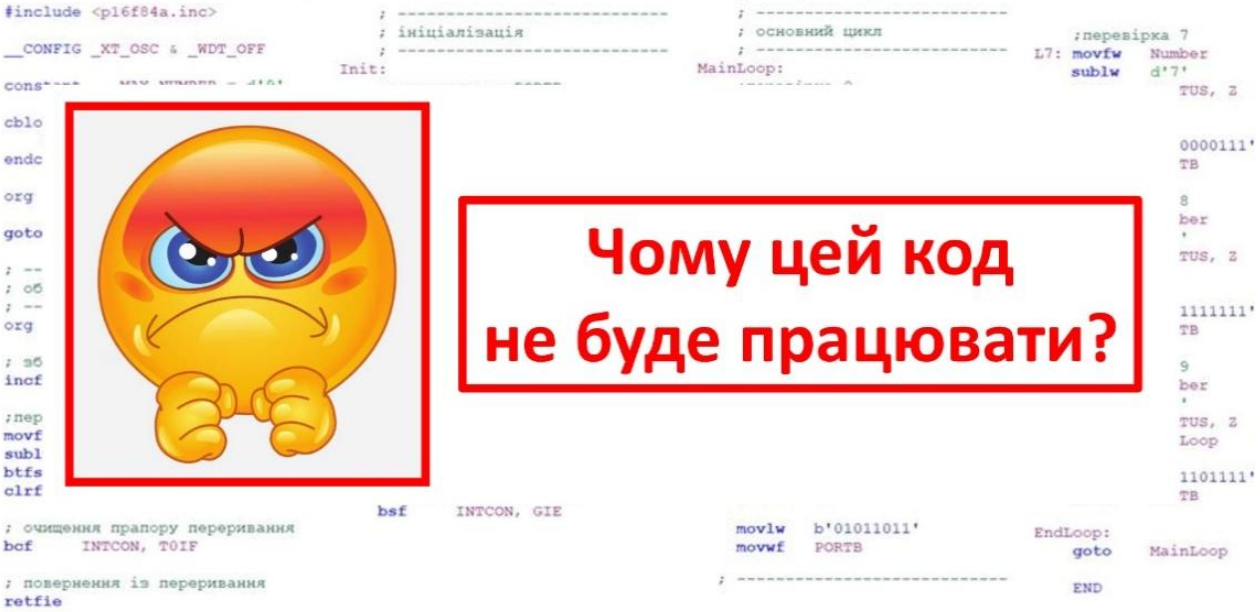


Наведено орієнтовну структуру програми для наступної практичної роботи. На початку виконується ініціалізація: з адреси org 0 здійснюється безумовний перехід у блок ініціалізації, де виконуються всі необхідні налаштування. Змінні обов’язково мають бути ініціалізовані, тобто для них повинні бути задані початкові значення.

Після ініціалізації виконується перевірка умови. У прикладі перевіряється значення на нуль, для чого використовується операція віднімання, хоча замість неї може бути застосоване виключне АБО. Якщо умова не виконується, здійснюється перехід на іншу ділянку коду. Якщо умова виконується, формуються сигнали, що відповідають поточному значенню.

Далі виконується перевірка, чи не перевищило значення допустиме максимальне. Значення збільшується, після чого порівнюється з максимальною константою. Якщо воно більше за допустиме, значення скидається командою очищення і записується початкове значення.

Варіант вихідного коду програми



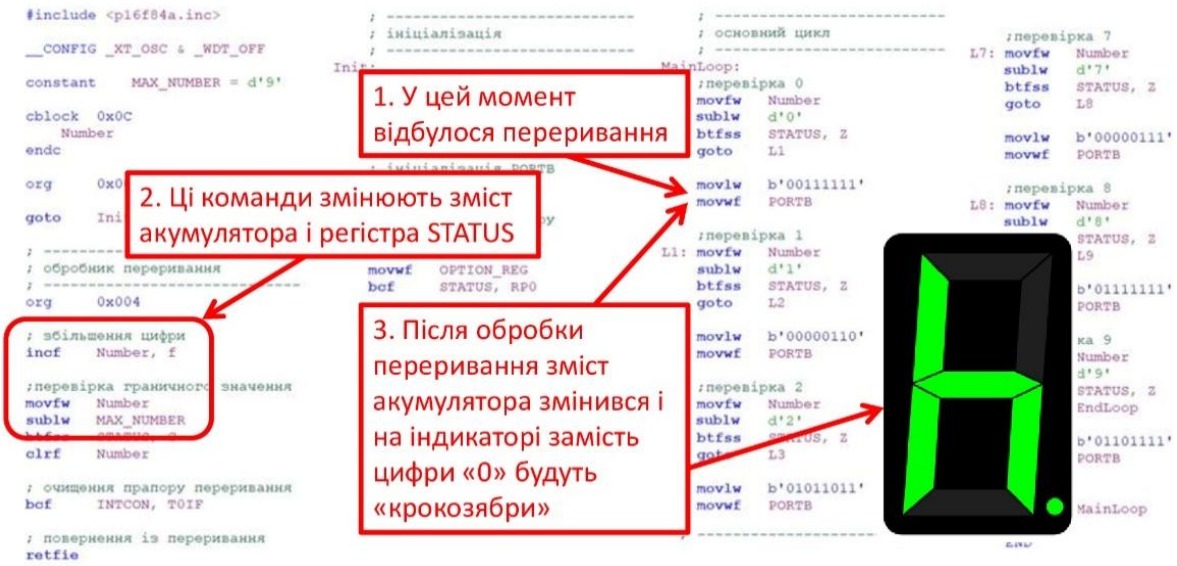
Показаний код у такому вигляді працювати не буде через один важливий нюанс. У програмі використовуються переривання, і під час входу в обробник переривання необхідно зберігати контекст.

Ситуація виглядає так. У основному коді формується відображення цифри, завантажується константа, що визначає стан сегментів, і ця константа розміщується в акумуляторі. Далі виконується копіювання значення з акумулятора в порт. У цей момент може виникнути переривання. Після переходу в обробник переривання виконуються операції зміни змінних, завантаження значень в акумулятор, арифметико-логічні операції, результати яких також зберігаються в акумуляторі.

Після завершення обробки переривання відбувається повернення до основної програми. У цей момент в акумуляторі вже знаходиться інше значення, ніж те, яке було до переривання. Внаслідок цього при подальшому використанні акумулятора замість очікуваної цифри на виході з'являються некоректні дані.

Переривання може виникнути на будь-якій команді. Виникає питання, що необхідно робити в такій ситуації.

Варіант вихідного коду програми, що не буде працювати



Перед обробкою переривання в першу чергу необхідно зберегти контекст. Контекст визначає стан процесора, який має бути відновлений після виходу з обробника переривання. Саме збереження контексту є обов'язковою першою дією під час входу в переривання.

У технічній документації на мікроконтролер наведений рекомендований приклад такого коду. Ключовим є не конкретна реалізація, а розуміння того, що збереження контексту має виконуватися одразу, до виконання будь-яких інших дій в обробнику переривання.

Збереження контексту

EXAMPLE 6-1: SAVING STATUS AND W REGISTERS IN RAM

```
PUSH    MOVWF   W_TEMP          ; Copy W to TEMP register,
        SWAPF   STATUS,     W   ; Swap status to be saved into W
        MOVWF   STATUS_TEMP     ; Save status to STATUS_TEMP register
ISR     :                       :
        :                       ; Interrupt Service Routine
        :                       ; should configure Bank as required
        :                       ;
POP     SWAPF   STATUS_TEMP,W   ; Swap nibbles in STATUS_TEMP register
                                ; and place result into W
        MOVWF   STATUS          ; Move W into STATUS register
                                ; (sets bank to original state)
        SWAPF   W_TEMP,     F   ; Swap nibbles in W_TEMP and place result in W_TEMP
        SWAPF   W_TEMP,     W   ; Swap nibbles in W_TEMP and place result into W
```

На самому початку обробки переривання потрібно зберегти в оперативній пам'яті поточний зміст акумулятора та регістра STATUS (інколи і інших регістрів) і відновити їх перед завершенням обробки переривання

Після входу в переривання контекст необхідно одразу зберегти. Далі виконується обробка переривання. Перед виконанням команди RETFIE контекст має бути відновлений, щоб після повернення в основний цикл програма продовжила роботу без збоїв.

Це останній важливий момент, на який слід звернути увагу під час написання практичної роботи.

Код обробника переривань, із збереженням контексту (який буде працювати)

```
#include <p16f84a.inc>

__CONFIG _XT_OSC & _WDT_OFF

constant    MAX_NUMBER = d'9'

cblock  0x0C
    Number
    STATUS_Temp
    W_Temp
endc

org     0x000

goto    Main
```

```
; ----------------------------
; обробник переривання
; ----------------------------
org     0x004

; збереження контексту
movwf   W_Temp
swapf   STATUS, w
movwf   STATUS_Temp

; збільшення цифри
incf    Number, f

;перевірка граничного значення
movfw   Number
sublw   MAX_NUMBER
btfss   STATUS, C
clrf    Number
```

```
; очищення прапору переривання
bcf     INTCON, T0IF

; відновлення контексту
swapf   STATUS_Temp, w
movwf   STATUS
swapf   W_Temp, f
swapf   W_Temp, w

; повернення із переривання
retfie
```

ПРОГРАМУВАННЯ ПРИСТРОЇВ ІНТЕРНЕТУ РЕЧЕЙ

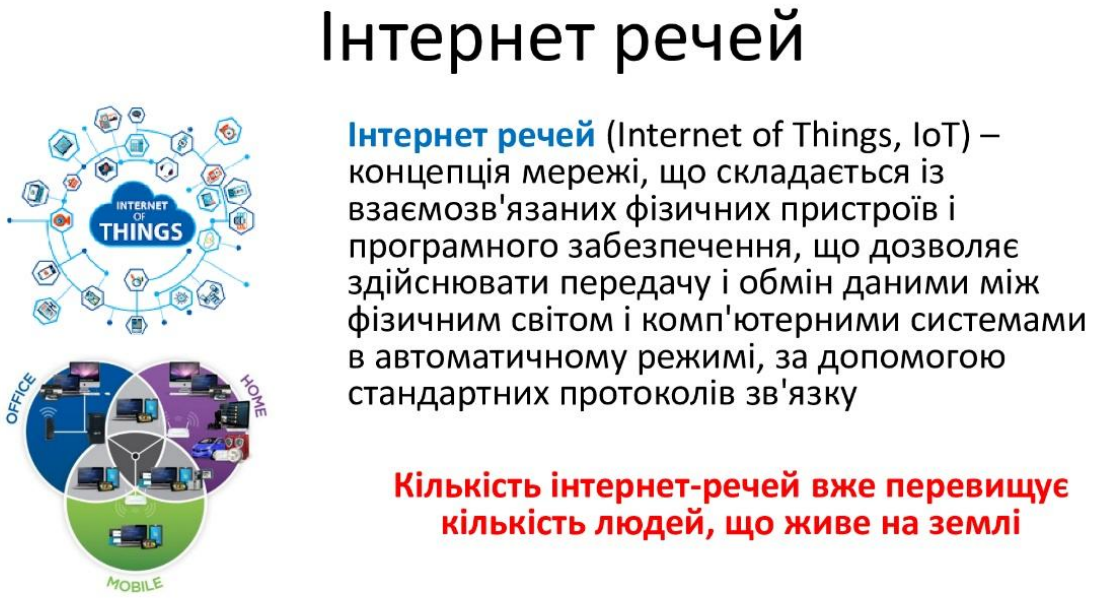
Протягом семестру було розглянуто базові поняття та механізми роботи мікроконтролерів, зокрема таймери, акумулятори, переривання та інші елементи низькорівневого програмування, що формують початкове уявлення про предмет.

Оскільки дисципліна охоплює також тематику інтернету речей, необхідно розглянути, як отримані знання пов'язані з цією сферою та як їх застосовувати на практиці. Подальша мета – сформувати загальне розуміння напрямків використання набутих знань і можливих шляхів розвитку після завершення курсу.

Інтернет речей – це концепція, за якої різноманітні пристрої можуть автономно, без безпосередньої участі людини, взаємодіяти через мережу Інтернет з іншими пристроями або сервісами та обмінюватися даними. Такі системи застосовуються у багатьох сферах, зокрема в системах «розумного» будинку та промисловому інтернеті речей.

Технологія перебуває у стадії активного розвитку: різні виробники створюють власні платформи та сервіси, формуються спільні підходи й архітектури. Кількість підключених до мережі пристроїв уже перевищує кількість населення Землі і продовжує зростати, що забезпечує довгострокову актуальність цього напрямку.

Подальший розгляд буде зосереджений не на сферах застосування, а на принципах створення пристроїв інтернету речей.



Більшість пристроїв інтернету речей створюється на основі мікроконтролерів. Хоча існують складні системи, що використовують повноцінні комп’ютери, найчастіше це невеликі пристрої, які виконують обмежене завдання, збирають дані та передають їх на сервер для подальшого накопичення й обробки.

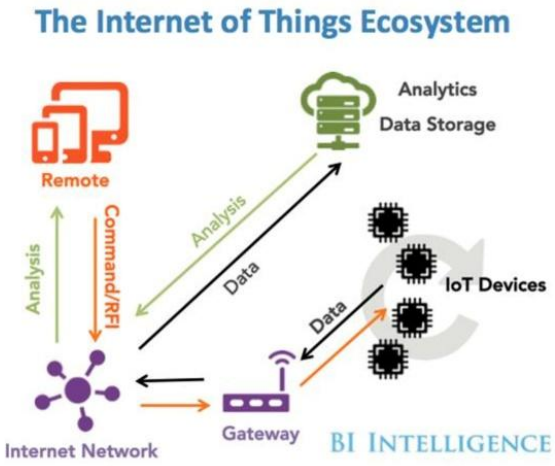
Застосунок інтернету речей принципово не відрізняється від інших мікроконтролерних систем, однак потребує особливої уваги до трьох аспектів – функціональності, безпеки даних і енергоспоживання. Саме ці характеристики визначають специфіку таких пристроїв.

У межах дисципліни мікроконтролер розглядається як кінцевий елемент екосистеми інтернету речей – джерело первинних даних. До нього підключаються датчики, що вимірюють параметри середовища, наприклад температуру, вологість, освітленість або інші показники. Отримані дані приводяться до формату, придатного для передавання, після чого надсилаються на сервер через канали зв’язку, де накопичуються та обробляються системами вищого рівня.

Типовий пристрій інтернету речей складається з трьох обов’язкових компонентів. Перший – мікроконтролер, що керує роботою системи та обробляє дані. Другий – модуль зв’язку, який забезпечує підключення до мережі Інтернет провідними або безпровідними технологіями, такими як Wi-Fi, Bluetooth або інші мережі. Третій – модуль шифрування, необхідний для захисту даних під час передавання відкритими каналами зв’язку.

Наявність цих трьох складових є обов’язковою умовою побудови пристрою інтернету речей, оскільки безпечний зв’язок і коректна обробка даних визначають його працездатність.

Екосистема Інтернету речей



Основні вимоги до пристроїв Інтернету речей

- Функціональність
- Безпека даних
- Енергоспоживання

Функціональність пристрою визначається можливостями мікроконтролера та його периферії. У курсі розглядався один із найпростіших мікроконтролерів – PIC16F84, який має мінімальний набір апаратних ресурсів. До периферії фактично належать лише порти введення-виведення, один простий таймер у вигляді регістра з автоматичним інкрементом та енергонезалежна пам’ять. На основі навіть такого мінімального набору можна реалізувати базові застосунки.

Сучасні мікроконтролери можуть містити значно ширший набір універсальних периферійних вузлів: порти введення-виведення, кілька таймерів, модулі широтно-імпульсної модуляції, сторожові таймери, вузли контролю напруги живлення, енергонезалежну пам’ять. Такий набір вважається типовим для більшості сучасних пристроїв.

Для керування стандартними індикаторами та іншими пристроями індикації також існують спеціалізовані апаратні модулі. Це дозволяє уникнути значних витрат процесорного часу на програмну реалізацію динамічної індикації або формування сигналів керування. Наприклад, для рідкокристалічних індикаторів, які потребують змінної напруги на сегментах, у багатьох мікроконтролерах передбачені окремі контролери дисплея, що автоматично формують необхідні сигнали, тоді як програмі потрібно лише передати дані для відображення.

Важливою складовою функціональності є комунікаційні можливості. Багато мікроконтролерів мають апаратну підтримку стандартних інтерфейсів обміну даними, зокрема I^2C, 1-Wire та інших. Ці інтерфейси широко використовуються і добре стандартизовані, тому їх часто реалізують апаратно для зменшення навантаження на процесор. Існують також мікроконтролери з підтримкою складніших протоколів зв’язку, включно з Ethernet, Bluetooth та іншими мережевими технологіями, де значна частина функціоналу вже реалізована на апаратному та програмному рівнях.

Окремо слід враховувати роботу з аналоговими сигналами. Для цього багато мікроконтролерів мають вбудовані аналогові вузли: компаратори, аналого-цифрові перетворювачі, підсилювачі, джерела опорної напруги та інші елементи, що дозволяють безпосередньо працювати з фізичними величинами без зовнішніх компонентів.

Забезпечення необхідної функціональності

Вузли загального призначення

- Порти введення-виведення загального призначення
- Таймери (із підтримкою ШІМ)
- Сторожові таймери
- Вузли контролю стану напруги живлення
- Енергонезалежна пам’ять

Інтерфейс користувача

- Контролери рідкокристалічних дисплеїв та індикаторів
- Контролери клавіатур

Комунікація та зв’язок

- Модулі стандартних інтерфейсів (UART, SPI, I^2C, CAN, USB, 1-Wire, Infrared тощо)
- Спеціалізовані програмні модулі (системи на кристалі) (Ethernet, Bluetooth, Wi-Fi, ZeegBee, тощо)

Вузли для роботи із аналоговими сигналами

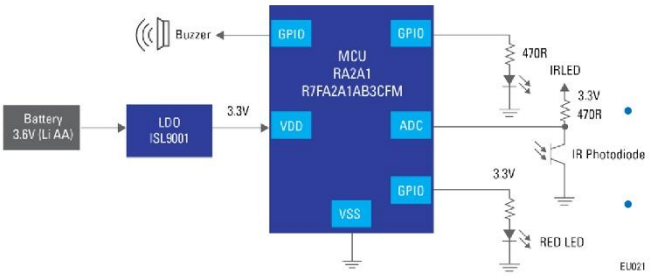
- Операційні підсилювачі
- Компаратори
- Аналого-цифрові перетворювачі
- Цифро-аналогові перетворювачі

Для забезпечення необхідної функціональності насамперед потрібно визначити вимоги до системи, після чого обрати мікроконтролер, який уже містить потрібний набір периферійних модулів.

Як приклад наведено простий пристрій – датчик диму. Його основою є оптична пара, що складається зі світлодіода та фототранзистора, які працюють в інфрачервоному діапазоні. Поява диму між ними змінює рівень освітлення фототранзистора, відповідно змінюється його опір і напруга на ньому. Ця напруга вимірюється аналого-цифровим перетворювачем мікроконтролера.

Додаткові елементи, такі як світлодіоди індикації та звуковий випромінювач із вбудованим генератором, підключаються до звичайних портів введення-виведення і керуються цифровими сигналами. У результаті формується повністю працездатний прикладний пристрій на основі мікроконтролера.

Датчик диму



Периферійні вузли мікроконтролера

- **GPIO** (General Purpose Input-Output) – порти введення-виведення зального призначення
- **ADC** (Analog-to-Digital Converter) – Аналого-цифровий перетворювач
- **LDO** (Low DropOut regulator) – стабілізатор із зниженим падінням напруги між входом та виходом

Більш складним прикладом пристрою інтернету речей є вимірювач рівня глюкози в крові. У такій системі мікроконтролер взаємодіє з тест-смужкою, що містить сенсорні елементи для вимірювання концентрації глюкози. Сигнали від датчиків надходять на вбудовані операційні підсилювачі, після чого подаються на 16-розрядний аналого-цифровий перетворювач для точного вимірювання.

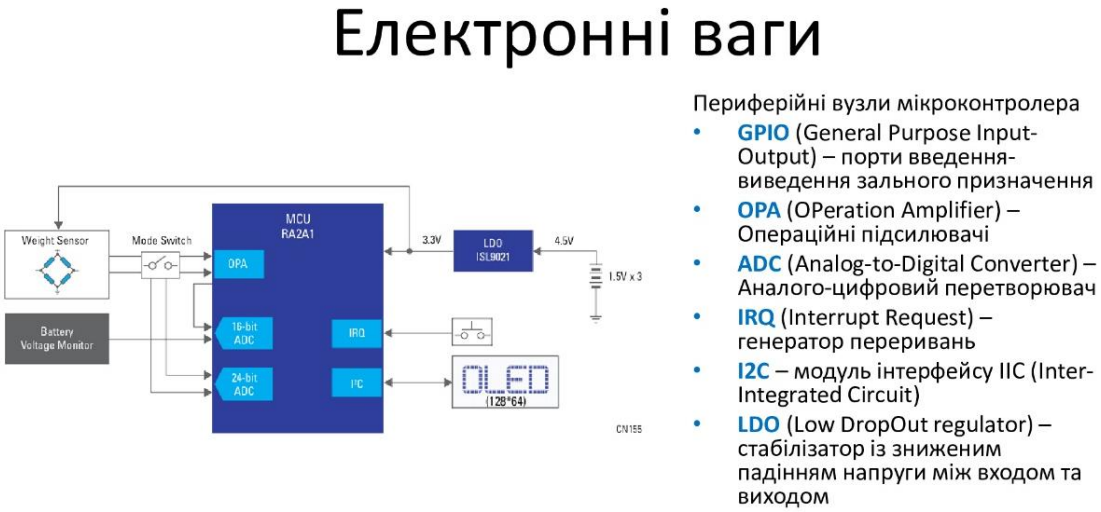
Пристрій також може містити аудіокодек, мікрофон і динамік, з’єднані з мікроконтролером через цифровий інтерфейс, світлодіодний індикатор для відображення інформації та звуковий сигналізатор, підключений до портів введення-виведення.

Ключовою ознакою пристрою інтернету речей є наявність модуля зв'язку. У цьому випадку використовується окремий модуль Bluetooth, підключений до мікроконтролера через інтерфейс UART. Завдяки цьому виміряні дані можуть передаватися на смартфон, комп'ютер або інший пристрій, що забезпечує віддалений доступ і обмін інформацією.

Для реалізації потрібної функціональності обирають мікроконтролер із необхідними вбудованими модулями.

Прикладом є електронні ваги. У них використовується матриця тензорезисторів, сигнали з якої подаються на операційні підсилювачі та далі на аналого-цифрові перетворювачі. Отримані дані обробляються математично, після чого обчислене значення ваги передається на індикатор.

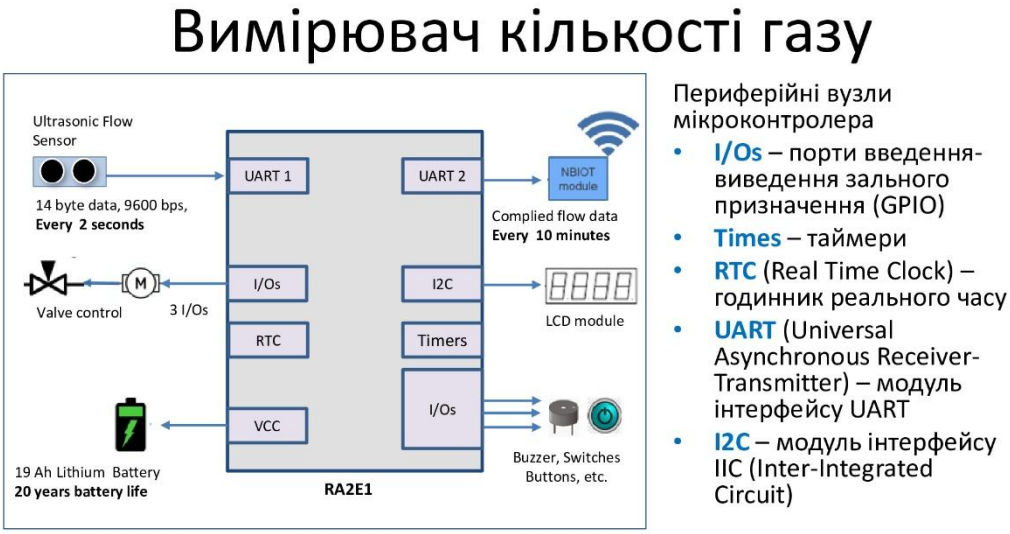
Індикатор на органічних світлодіодах є окремим пристроєм із власним мікроконтролером, тому підключається не напряму до портів введення-виведення, а через інтерфейс I²C. Це дозволяє замість великої кількості проводів використовувати лише два сигнальні провідники інтерфейсу, передаючи команди відображення даних.



Ще одним прикладом є газоаналізатор – вимірювач концентрації газів. Такий пристрій також містить датчики, елементи введення-виведення та модулі зв'язку, які разом утворюють завершену систему.

Розробник повинен розуміти принцип роботи датчиків на фізичному рівні, оскільки вони можуть мати різні типи вихідних сигналів. Одні датчики формують аналогову напругу, пропорційну вимірюваній величині, інші є повністю автономними цифровими пристроями. Наприклад, ультразвукові або інші складні сенсори часто містять власний мікроконтролер, який виконує первинну обробку, калібрування та видає вже готові цифрові дані.

Використання таких готових цифрових датчиків є сучасною тенденцією розробки. Замість створення систем «з нуля» зазвичай застосовують максимально завершені модулі, що скорочує час розробки та підвищує надійність кінцевого пристрою.



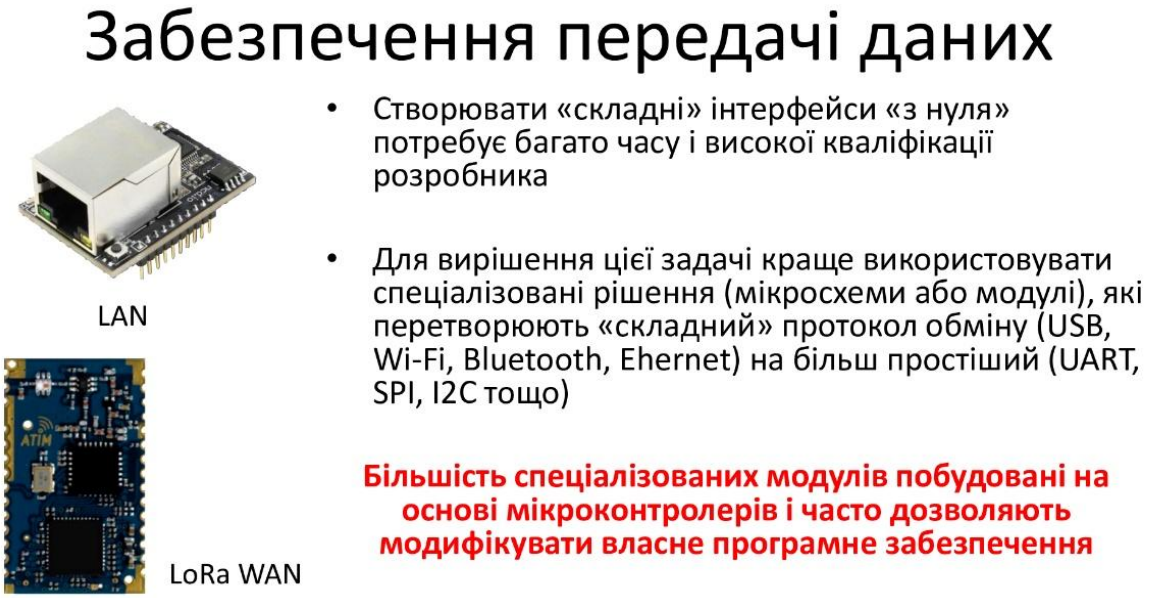
Прості інтерфейси зв’язку, такі як UART, I²C або SPI, належать до базових і можуть бути реалізовані або зрозумілі за відносно короткий час, у тому числі програмно. Натомість складні безпровідні інтерфейси, зокрема Wi-Fi або Bluetooth, значно важчі для самостійної реалізації. Їх створення з нуля потребує великих витрат часу на розробку та тестування і може зайняти від місяців до років навіть за наявності досвіду.

Тому на практиці для підключення до таких мереж використовують готові модулі. Вони фактично є окремими мікроконтролерними системами з власним програмним забезпеченням, радіочастотною частиною та часто вбудованими антенами. Основна їх функція – перетворення складного мережевого інтерфейсу у простіший для основного мікроконтролера, наприклад у UART або SPI.

У такому випадку головний контролер передає модулю дані у вигляді байтів, а модуль сам виконує всі операції мережевого обміну, включно з автентифікацією та протоколами зв’язку. Використання таких рішень є поширеним завдяки їх доступності, невеликій вартості та компактності. Це дозволяє додати складні функції зв’язку до пристрою без необхідності самостійної реалізації відповідних технологій.

Існує широкий спектр готових модулів зв’язку – від модулів для провідного підключення до мережі до мобільних GPRS-модулів. Останні фактично є повноцінними стільниковими пристроями: до них підключається GSM-антена , встановлюється SIM-карта, після чого модуль автоматично реєструється в мережі оператора та може передавати дані через мобільний зв’язок за допомогою пакетних протоколів.

Завдяки цьому відпадає потреба самостійно розробляти складні системи зв’язку. Для додавання необхідної функції достатньо обрати готовий модуль, інтегрувати його в пристрій і використовувати через стандартний інтерфейс.



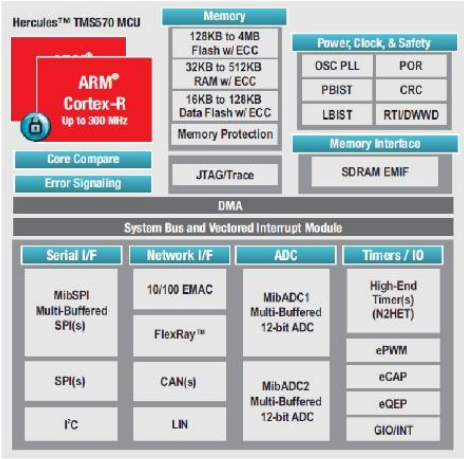
Безпеку в системах на мікроконтролерах необхідно розглядати на двох рівнях – апаратному і програмному. Сфера застосування мікроконтролерів значно ширша за інтернет речей і включає системи, від яких безпосередньо залежить життя людей: керування ліфтами, автомобілями, літальними апаратами та іншими критичними об’єктами. У таких випадках відмова системи може мати катастрофічні наслідки, тому вимоги до надійності суттєво вищі.

Для критичних застосувань існують спеціалізовані мікроконтролери підвищеної надійності, наприклад сімейства, призначені для функціональної безпеки. Їх особливістю є використання двох ідентичних обчислювальних ядер, які паралельно виконують ті самі операції. Результати порівнюються спеціальним апаратним модулем. Якщо виявляється розбіжність, що може бути спричинена фізичним пошкодженням кристала, електромагнітною завадою або проблемами живлення, система формує аварійний сигнал, ініціює переривання та переходить у безпечний режим.

Додатково захищаються дані у всіх типах пам’яті та на внутрішніх шинах. Для передаваних даних апаратно обчислюються контрольні суми, які зберігаються разом із ними і використовуються для перевірки цілісності. Оперативна пам’ять також має механізми контролю коректності, що дозволяють виявляти спонтанні зміни даних. Окрім цього, застосовуються інші багаторівневі технології, спрямовані на мінімізацію ймовірності відмови системи.

Таким чином, апаратна безпека забезпечується спеціальними архітектурними рішеннями, характерними для систем, де критично важлива безвідмовна робота.

Безпека на апаратному рівні



- Використання двох ядер, що виконують однакові арифметико-логічні операції
- Додавання контрольної суми до даних, що зберігаються у всіх видах пам'яті
- Додавання контрольної суми до даних, що передаються по внутрішнім шинам

Безпека на апаратному рівні використовуються в застосунках, від яких залежить безпека людей (ліфти, автомобілі, авіація, космічна техніка, тощо)

У пристроях інтернету речей основну роль відіграє програмна безпека. Оскільки такі пристрої підключені до мережі Інтернет, доступ до них можуть отримувати не лише власники, а й сторонні особи. Несанкціоноване керування побутовими пристроями може бути лише незручністю, але компрометація персональних даних може мати серйозні наслідки.

Захист реалізується на кількох рівнях. На рівні користувача застосовуються механізми автентифікації – логіни, паролі та розмежування прав доступу, наприклад ролі адміністратора і звичайного користувача. Ці механізми реалізуються програмно.

Окремо необхідно забезпечити безпечний канал передавання даних. Для цього широко використовуються апаратні засоби шифрування, оскільки виконання криптографічних операцій лише силами мікроконтролера може суттєво навантажувати процесор і займати значну частину обчислювального часу.

Ризики пристроїв Інтернету речу

Основні ризики

- Можливість витоку персональних даних
- Можливість стороннього контролю вашими пристроями

Методи захисту

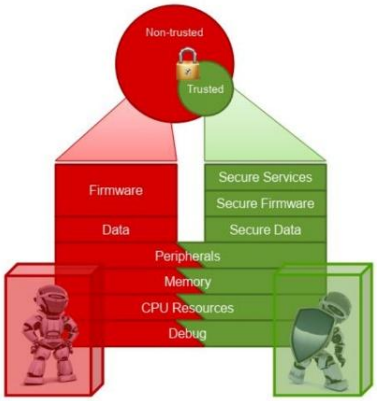
- Апаратні (шифрування даних)
- Організаційні (збереження паролів, багаторівневий рівень доступу)

Для забезпечення програмної безпеки ресурси мікроконтролера умовно поділяють на дві області – незахищену та довірену. Під ресурсами маються на увазі всі компоненти системи: оперативна пам'ять, інтерфейси, порти введення-виведення та інші апаратні можливості.

У довіреній зоні розміщується операційна система реального часу, яка контролює доступ до ресурсів і надає дозволи прикладним програмам через визначені інтерфейси. Таким чином формується багаторівнева система захисту, де прикладний код не має прямого доступу до критичних компонентів.

Реалізація такого підходу потребує підтримки з боку апаратури. Мікроконтролер повинен мати спеціальні модулі, регістри та механізми ізоляції, що забезпечують поділ ресурсів і контроль доступу. Такі рішення широко застосовуються на практиці в системах, де важлива інформаційна безпека.

Розділення мікроконтролера на дві зони

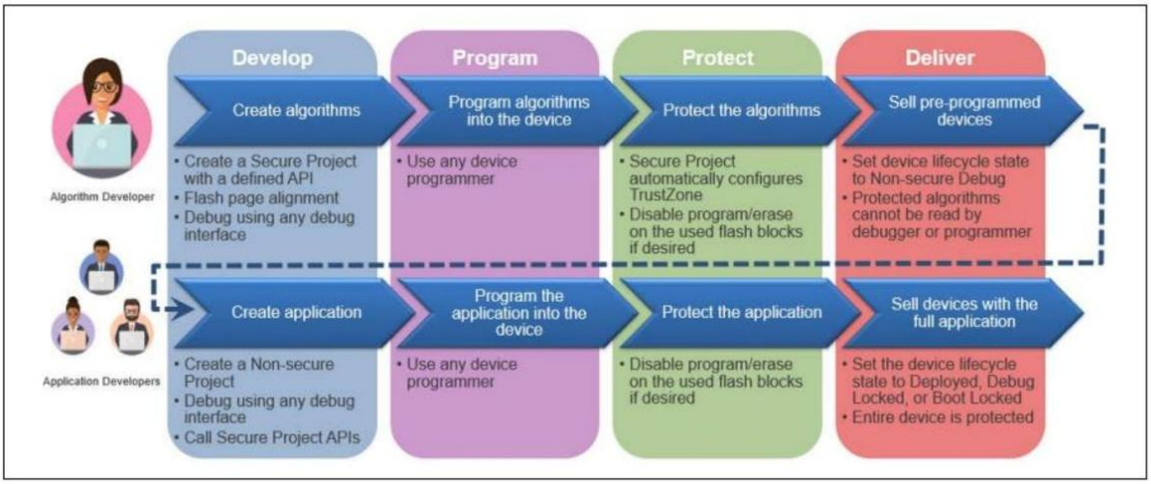


- Розділення мікроконтролера на дві зоні: звичайну та довірену (Trust Zone)
- Доступ до внутрішніх ресурсів мікроконтролера, у тому числі і до периферійних пристроїв і портів введення-виведення, забезпечується тільки через захищену область, через функції API (Application Programming Interface), що надаються та контролюються операційною системою

У результаті побудови захищеної системи одного програміста вже недостатньо. Формується розподіл ролей: окрема група розробників має повний доступ до внутрішніх механізмів, знає архітектуру операційної системи, принципи її роботи та реалізує довірену частину системи. Ця частина зазвичай є закритою і не розголошується.

Інші розробники створюють прикладне програмне забезпечення, використовуючи надані інтерфейси доступу, але не мають прямого доступу до захищених механізмів. Таким чином формується командна розробка, де різні спеціалісти відповідають за різні рівні системи.

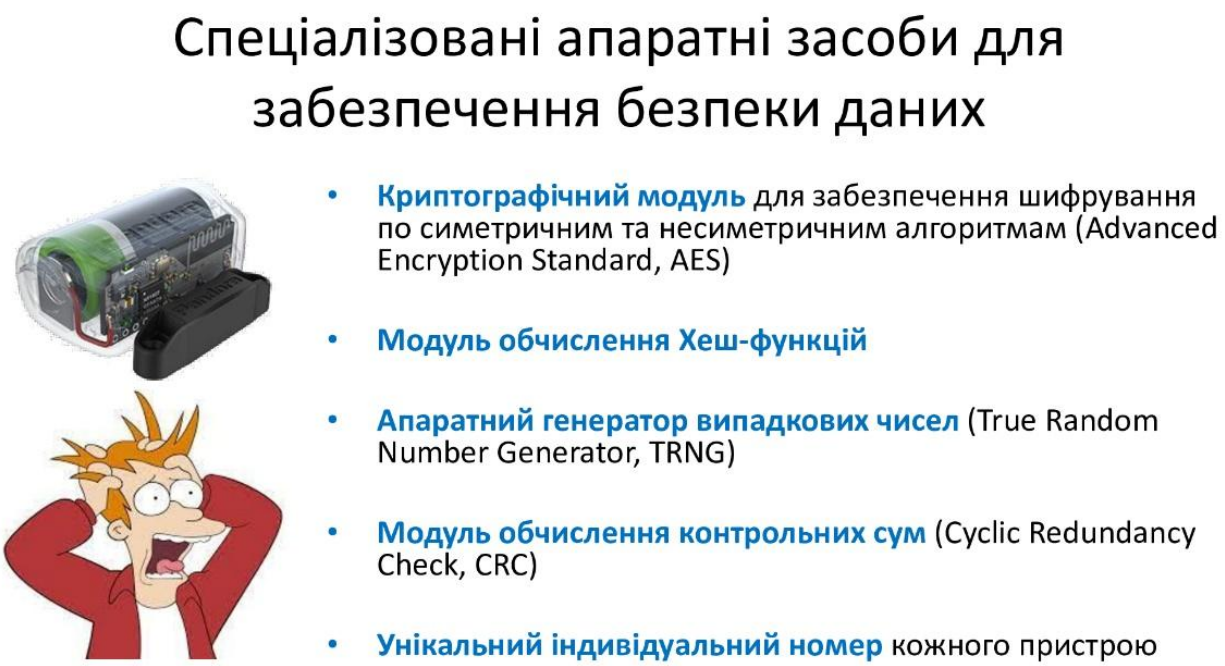
Процес розробки програмного забезпечення при використанні довірених зон



Для пристроїв інтернету речей забезпечення захисту є обов’язковим. Без нього система може функціонувати, але з високою ймовірністю стане вразливою до несанкціонованого доступу. Тому в сучасні мікроконтролери інтегруються спеціалізовані апаратні засоби безпеки: криптографічні модулі, блоки обчислення хеш-функцій і контрольних сум, апаратні генератори випадкових чисел на основі фізичного шуму, а також унікальні ідентифікатори пристрою, що не повторюються.

Прикладом є безпровідний датчик відкриття дверей або вікна. Він містить магніт і датчик Холла, що реагує на магнітне поле, мікроконтролер, який аналізує стан сенсора, та модуль зв’язку для передавання інформації на сервер або іншу систему. Такі датчики застосовуються в охоронних системах, системах контролю доступу або моніторингу стану об’єктів, наприклад дверей автомобіля. Пристрій зазвичай живиться від батареї, а передавання даних здійснюється безпровідним каналом.

Навіть якщо потрібно передати лише один біт інформації – стан «відкрито» або «закрито», – необхідно реалізувати повний комплекс криптографічного захисту. Це значно ускладнює систему і створює великий обсяг роботи для розробника, оскільки безпечна передача навіть мінімальних даних потребує складних механізмів автентифікації та шифрування.



Реалізація криптографічних алгоритмів є складним і трудомістким завданням. Навіть за наявності готових програмних рішень значний час витрачається на перевірку коректності, надійності та стійкості до атак, причому гарантований результат не завжди очевидний.

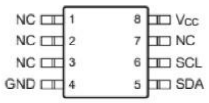
Тому виробники пропонують спеціалізовані апаратні мікросхеми безпеки, які реалізують необхідні криптографічні функції всередині себе. Такі чипи забезпечують повний рівень захисту, необхідний для пристроїв інтернету речей, і випускаються у різних корпусах. Зазвичай вони мають мінімальну кількість виводів – живлення та інтерфейс обміну даними, наприклад I^2C.

Принцип роботи полягає в тому, що мікроконтролер передає до мікросхеми відкриті дані, після чого вона виконує всі криптографічні операції і повертає вже зашифрований результат. Усі алгоритми шифрування, генерація ключів та інші процедури виконуються всередині чипа, без необхідності реалізації цих механізмів у основному програмному коді.

Для використання достатньо вивчити технічну документацію, команди керування та протокол обміну, після чого інтегрувати мікросхему у систему.

ATECC608 – криптографічний сопроцесор

8-lead SOIC
(Top View)



Cryptographic Co-Processor with Secure Hardware-Based Key Storage:
- Protected storage for up to 16 keys, certificates or data

Hardware Support for Asymmetric Sign, Verify, Key Agreement:
- ECDSA: FIPS186-3 Elliptic Curve Digital Signature
- ECDH: FIPS SP800-56A Elliptic Curve Diffie-Hellman
- NIST Standard P256 Elliptic Curve Support

Hardware Support for Symmetric Algorithms:
- SHA-256 & HMAC Hash including off-chip context save/restore
- AES-128: Encrypt/Decrypt, Galois Field Multiply for GCM

Networking Key Management Support:
- Turnkey PRF/HKDF calculation for TLS 1.2 & 1.3
- Ephemeral key generation and key agreement in SRAM
- Small message encryption with keys entirely protected •

Спеціалізовані криптографічні мікросхеми випускають різні виробники, зокрема компанія Microchip. У неї існує платформа довірених рішень, у межах якої такі чипи постачаються з різним рівнем попереднього налаштування.

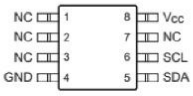
У варіанті Trust&GO надаються вже запрограмовані мікросхеми з унікальними ідентифікаторами та криптографічними ключами, готові до роботи з популярними хмарними платформами інтернету речей. Виробник забезпечує унікальність ключів і серійних номерів, а замовлення можливе навіть для малих партій.

Пакет Trust Flex передбачає напівготові рішення: апаратна частина вже налаштована, встановлено стандартні ключі, але подальше конфігурування виконується розробником самостійно за наданою документацією.

Пакет Trust Custom постачається як чиста апаратна основа без попереднього програмування. У цьому випадку всі механізми безпеки, ключі та сертифікати розробляються і завантажуються самостійно. Через високу складність і вартість таких робіт цей варіант зазвичай використовується лише у великосерійному виробництві з великими обсягами замовлення.

ATECC608 – криптографічний сопроцесор

8-lead SOIC
(Top View)



Варіанти замовлення мікросхем
- **Trust & GO** – повністю готова мікросхема з унікальними ідентифікаційними даними, розрахована на роботу з конкретними сервісами (AWS IoT, Microsoft Azure IoT Hub, Google IoT, LoRa, тощо) (заказ не менше 10 мікросхем)
- **Trust Flex** – мікросхеми із встановленими загальними ключами і сертифікатами, які можна замінити самостійно (заказ не менше 2 000 мікросхем)
- **Trust Custom** – «чисті» мікросхеми – уся «критична» частина коду залишається в компанії (замовлення від 4000 мікросхем)

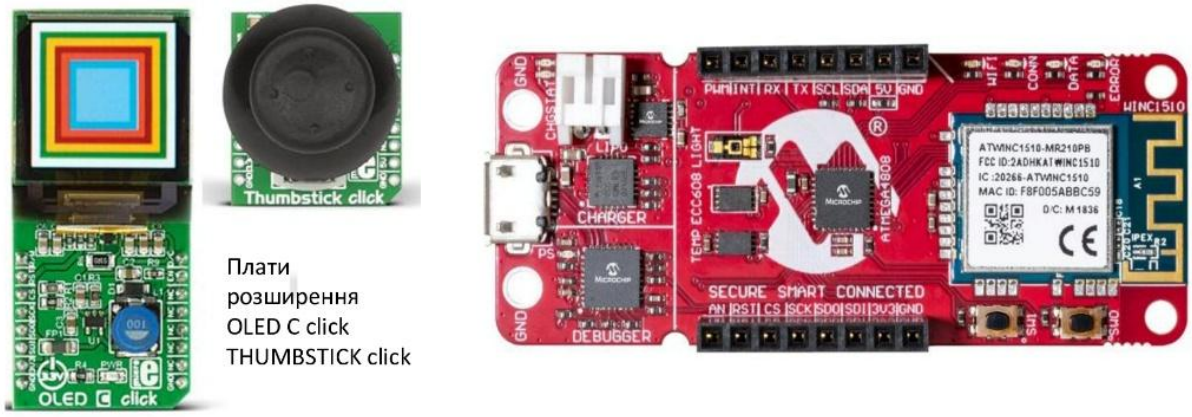
Створення пристроїв інтернету речей з апаратної точки зору часто зводиться до використання готових демонстраційних або оціночних плат. Вони дорожчі за окремі мікроконтролери, але містять повністю зібрану та перевірену апаратну основу.

На такій платі зазвичай встановлено мікроконтролер, модуль зв'язку (Bluetooth, Wi-Fi або інший), апаратний криптографічний чип, а також інтегрований програматор для завантаження прошивки. Додатково можуть бути вбудовані датчики, наприклад температури чи освітленості, і контролер заряджання акумулятора, що дозволяє живити пристрій від літій-іонної батареї.

Плата фактично є заготовкою готового IoT-пристрою. Для розширення функціональності використовуються спеціальні роз'єми, до яких підключаються додаткові модулі – дисплеї, органи керування, сенсори та інші периферійні пристрої. Існують великі набори таких плат розширення з різними функціями, що підключаються через стандартні інтерфейси та часто мають власні контролери.

Такий підхід дозволяє швидко зібрати прототип за принципом конструктора без розробки апаратури з нуля. Хоча вартість рішень досить висока, вони економлять значний час і зменшують ризик помилок, оскільки всі компоненти вже протестовані. Оціночні плати дають можливість швидко визначити доцільність подальшої розробки та перевірити ідею перед створенням власного пристрою.

Апаратна платформа для створення Інтернет-речей (EV15R70A або EV54Y39A)



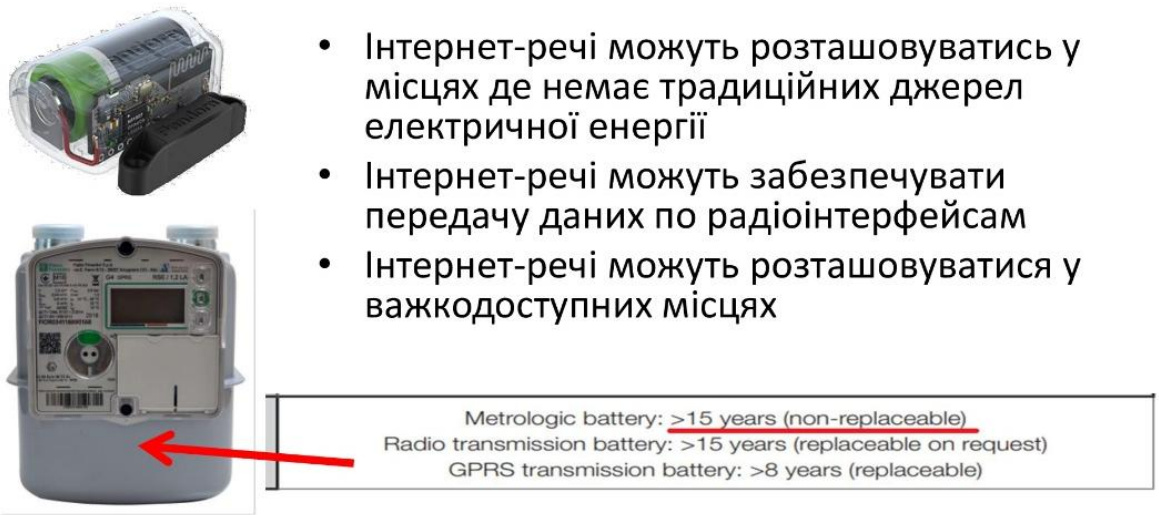
На апаратному та програмному рівнях створення пристроїв інтернету речей може бути відносно швидким завдяки готовим компонентам і платформам. Однак важливою особливістю таких систем є енергоспоживання.

Багато IoT-пристроїв не мають постійного або необмеженого джерела живлення. Наприклад, безпровідний датчик відкриття дверей встановлюється у важкодоступному місці саме для того, щоб уникнути прокладання кабелів. Подібні пристрої працюють автономно від батареї.

Іншим прикладом є газовий лічильник, що живиться від однієї батареї протягом багатьох років, іноді до десятиліття і більше. Досягнення такого ресурсу можливе лише за умови надзвичайно низького середнього споживання енергії.

Тому під час розробки пристроїв, що працюють від батарей, необхідно враховувати специфіку джерела живлення та оптимізувати роботу системи з урахуванням обмеженого енергетичного ресурсу.

Причини зменшення енергоспоживання

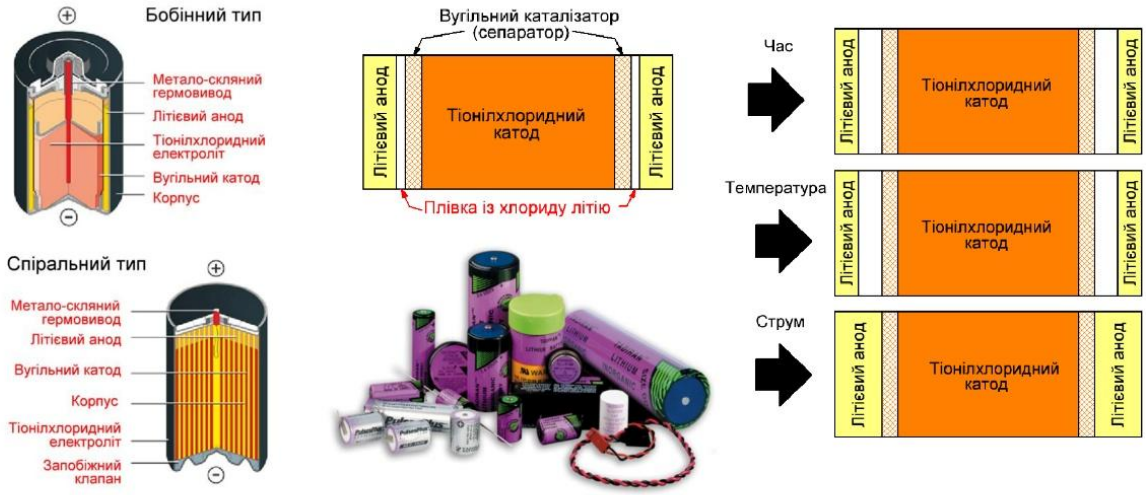


Тривалий термін роботи від однієї батареї досягається переважно при використанні літієвих первинних елементів живлення. Інші типи батарей не забезпечують необхідної питомої енергії для багаторічної автономної роботи, якщо не враховувати спеціалізовані або екзотичні джерела живлення. Йдеться саме про одноразові літієві батареї, а не акумулятори.

Літієві елементи мають характерну особливість: під час контакту літієвого електрода з активною масою іншого електрода на поверхні літію утворюється тонка пасивуюча плівка. Ця плівка перешкоджає проходженню електричного струму і є результатом природної хімічної реакції, яку неможливо повністю зупинити. Швидкість її утворення та товщина залежать від часу зберігання, температури та інших умов, тому навіть батареї з однієї партії можуть поводитися по-різному в різних пристроях.

Під час розряду батареї відбуваються паралельні процеси – плівка одночасно утворюється і частково руйнується. Коли струм споживання є, її вплив зменшується, а під час тривалого простою формування плівки переважає, що погіршує провідність і здатність батареї віддавати струм. Це явище необхідно враховувати при проєктуванні автономних IoT-пристроїв.

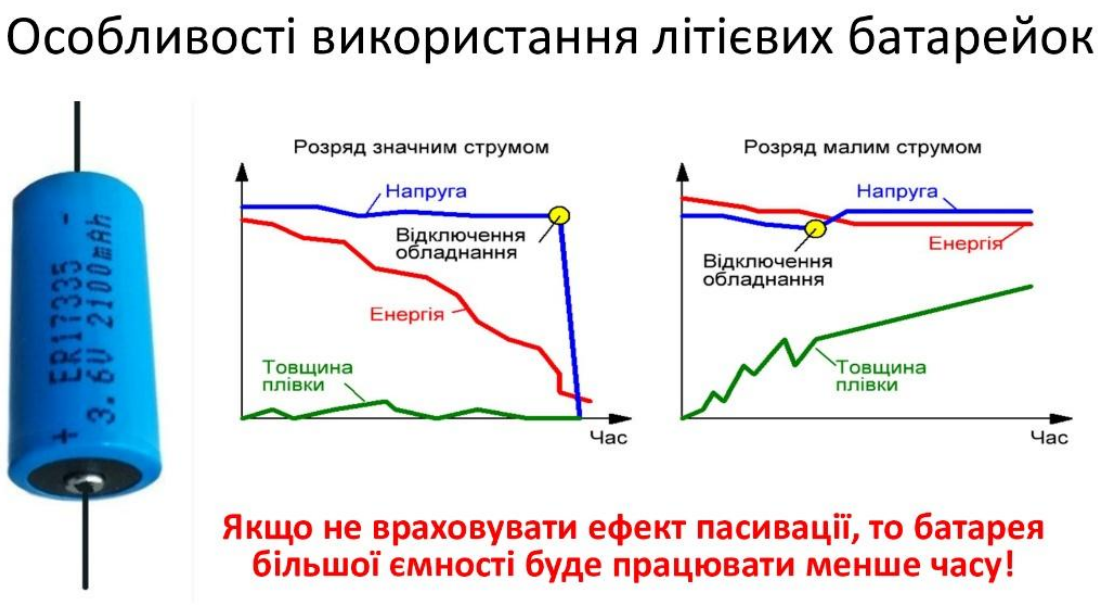
Літієві батарейки



Особливістю пристроїв інтернету речей є використання бездротових інтерфейсів зв'язку – Bluetooth, Wi-Fi, GPRS та інших. Передавання даних радіоканалом потребує значних витрат енергії, тому споживаний струм може різко змінюватися залежно від режиму роботи.

У режимі очікування або сну струм споживання може становити мікроампери або частки міліампера. Під час передавання даних він зростає до десятків або сотень міліампер, тобто змінюється на кілька порядків і робить це дуже швидко. Такі імпульсні навантаження потребують відповідних стабілізаторів живлення, здатних коректно працювати при різких змінах струму.

Оскільки активна передача відбувається відносно рідко, більшу частину часу пристрої інтернету речей перебувають у режимах енергозбереження або сну, де споживання мінімальне. Саме чергування коротких енергоємних сеансів зв'язку з тривалими періодами низького споживання дозволяє досягати тривалої автономної роботи.



Якщо не враховувати особливості літієвих батарей, автономний пристрій інтернету речей може працювати некоректно. Типовий режим роботи – тривалий сон із мінімальним споживанням, періодичне пробудження, збір даних і коротка передача по радіоканалу з різким зростанням струму. Під час активної роботи пасивуюча плівка на електроді частково руйнується, і батарея здатна віддавати великий струм.

Проблема виникає після тривалого простою. Якщо пристрій місяцями не використовується, батарея майже не розряджається, але пасивуюча плівка продовжує наростати. У момент наступного запуску батарея може виявитися нездатною видати необхідний імпульсний струм для передавання даних, хоча запас енергії в ній ще є. У результаті система може аварійно вимкнутися або повідомити про несправність живлення.

Щоб уникнути цього, програмне забезпечення повинно перед початком роботи перевіряти стан джерела живлення і за потреби виконувати депасивацію батареї – короткочасне навантаження, яке руйнує плівку і відновлює здатність віддавати струм. Лише після цього можна переходити до основних операцій, зокрема передачі даних. Якщо депасивація не дала результату, система повинна сформувати повідомлення про необхідність заміни батареї.

Без урахування цього механізму неможливо забезпечити багаторічну автономну роботу пристрою від однієї батареї.

Особливості живлення від літієвих батарейок

При створенні пристроїв, що живляться від літієвих батарейок програміст мікроконтролерів

- Контролює стан батареї
- Враховує ефекти пасивації
- Забезпечує процес депасивації

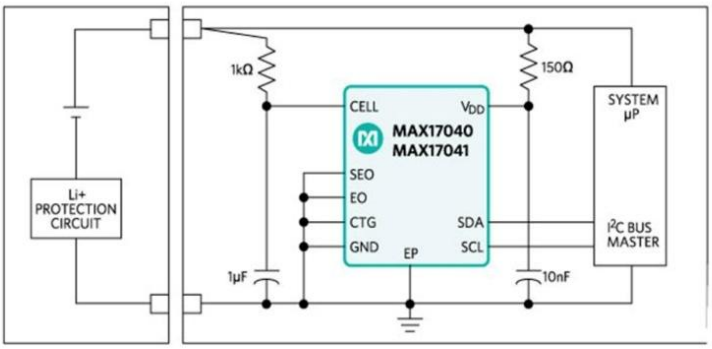
При створенні застосунків, що працюють від батарей, програміст повинен знати кількість енергії в батареї (акумуляторі) і розраховувати орієнтовний час її використання

Для автономних пристроїв, що живляться від батарей або акумуляторів, критично важливо знати фактичний запас енергії. Це особливо актуально для рухомих систем, наприклад роботів, квадрокоптерів і дронів, які повинні не лише виконати завдання, а й мати достатній ресурс для повернення.

Для контролю стану джерела живлення використовуються спеціалізовані мікросхеми – монітори живлення. Вони призначені для різних типів батарей і акумуляторів та відстежують енергію, що надходить під час заряджання і витрачається під час роботи. Такі чипи виконують складні обчислення, використовуючи математичні моделі, які враховують, зокрема, температурні впливи та характеристики елемента живлення.

Монітор підключається до мікроконтролера через стандартний інтерфейс, наприклад I²C, і передає вже оброблені дані. У результаті система отримує готову інформацію про залишок енергії – у джоулях, ват-годинах, відсотках або прогнозованому часі роботи. Це дозволяє коректно керувати автономністю пристрою та запобігати аварійним ситуаціям через розряд батареї.

Вимірювачі кількості енергії MAX17040/MAX17041



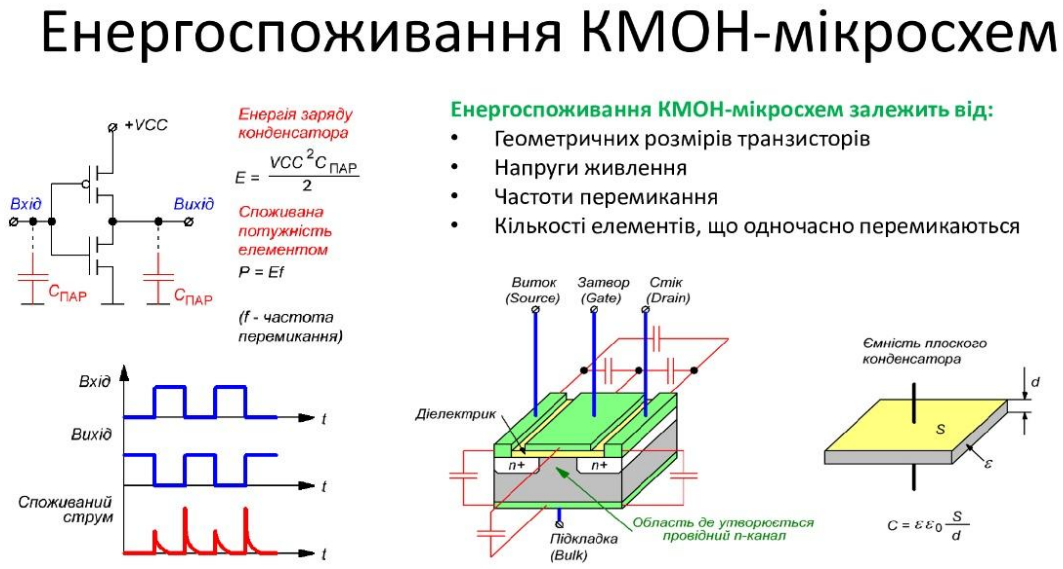
Вимірювачі кількості енергії літієвих батарей та акумуляторів роблять обчислення на основі патентованої математичної моделі ModelGauge з урахуванням великої кількості параметрів, у тому числі і температури

Для автономних систем необхідно постійно контролювати енергоспоживання і розуміти, від чого воно залежить. Мікроконтролери виготовляються на основі КМОН-транзисторів (комплементарних МОП-структур). Найпростіший логічний елемент, наприклад інвертор, складається з двох таких транзисторів.

У статичному стані, коли елемент не перемикається, струм практично не споживається, оскільки відсутній шлях протікання струму. Якщо зупинити тактовий генератор мікроконтролера, енергоспоживання може зменшитися майже до нуля, при цьому дані в оперативній пам'яті зберігаються. Основні витрати енергії виникають під час перемикання логічних елементів – на фронтах і спадах тактового сигналу.

Споживаний струм має вигляд коротких імпульсів, пов'язаних із заряджанням і розряджанням паразитних ємностей транзисторів. Кожен транзистор через свої геометричні розміри утворює мікроконденсатор, і саме на перезаряд цих ємностей витрачається енергія.

Енергоспоживання мікроконтролера визначається чотирма основними факторами: геометричними розмірами елементів (що впливають на ємність), напругою живлення (енергія пропорційна квадрату напруги), частотою перемикань і кількістю елементів, які одночасно змінюють стан. Останній фактор значною мірою залежить від програмного коду, тому програміст також впливає на загальне споживання енергії.



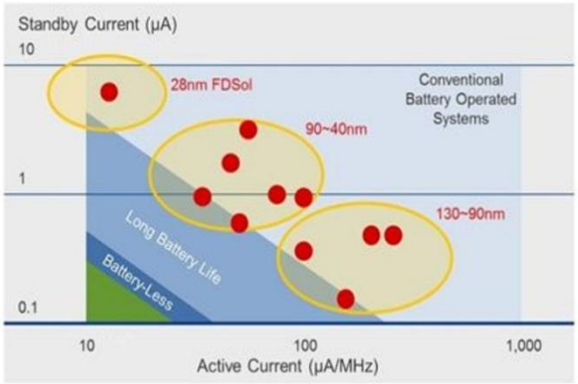
Для застосунків інтернету речей важливо враховувати не лише енергоспоживання під час активної роботи, а й струм у режимі сну, де пристрій перебуває більшу частину часу. Саме цей режим часто визначає загальний ресурс батареї.

Зменшення розмірів транзисторів дійсно знижує енергоспоживання під час перемикання в активному режимі. Однак у неактивному стані виникає протилежний ефект – зростають струми витоку. Через надзвичайно малі розміри транзисторів з'являється паразитний струм, що «просочується» через структуру навіть без перемикання. Хоча величина цього струму дуже мала для одного елемента, у мікроконтролері містяться мільйони транзисторів, тому сумарні втрати можуть бути значними.

Розміри транзисторів визначаються технологічним процесом виготовлення (наприклад, десятки нанометрів). Для процесорів загального призначення зменшення техпроцесу майже завжди означає менше енергоспоживання. Для IoT-пристроїв залежність неоднозначна: менші транзистори економніші під час роботи, але споживають більше через витоки у режимі сну.

Тому при виборі мікроконтролера потрібно враховувати характер роботи застосунку. Якщо пристрій більшу частину часу активний, доцільний сучасний малий техпроцес. Якщо ж він переважно перебуває у сплячому режимі, вигідніше використовувати мікроконтролер із більшим техпроцесом, який може мати менші струми витоку і навіть нижчу вартість.

Залежність енергоспоживання від розміру технологічного процесу



Зменшення розміру техпроцесу призводить до

- Зменшення струму, споживаному в активних режимах
- Збільшення струмів витоків у режимах зниженого енергоспоживання

Програміст повинен розуміти у яких режимах буде знаходитись застосунок більшу частину часу і обрати мікроконтролер із відповідним розміром техпроцесу

Для зменшення споживання енергії використовується режим зниженого енергоспоживання – режим сну. У мікроконтролері PIC16F84 він активується спеціальною асемблерною командою SLEEP.

У цьому режимі зупиняється тактовий генератор, припиняється виконання команд і більшість внутрішніх процесів. Залишається активним сторожовий таймер (Watchdog Timer), можуть виконуватися окремі операції з енергонезалежною пам’яттю, а також працюють певні джерела переривань, здатні вивести мікроконтролер зі стану сну.

Перехід у режим сну здійснюється програмно, тоді як вихід із нього зазвичай ініціюється апаратною подією – перериванням або сигналом скидання на вході RESET.

Режим сну у мікроконтролерах PIC16F84A

6.11 Power-down Mode (SLEEP)

A device may be powered down (SLEEP) and later powered up (wake-up from SLEEP).

6.11.1 SLEEP

The Power-down mode is entered by executing the SLEEP instruction.

If enabled, the Watchdog Timer is cleared (but keeps running), the $\overline{PD}$ bit (STATUS<3>) is cleared, the $\overline{TO}$ bit (STATUS<4>) is set, and the oscillator driver is turned off. The I/O ports maintain the status they had before the SLEEP instruction was executed (driving high, low, or hi-impedance).

For the lowest current consumption in SLEEP mode, place all I/O pins at either VDD or VSS, with no external circuitry drawing current from the I/O pins, and disable external clocks. I/O pins that are hi-impedance inputs should be pulled high or low externally to avoid switching currents caused by floating inputs. The T0CKI input should also be at VDD or VSS. The contribution from on-chip pull-ups on PORTB should be considered.

The $\overline{MCLR}$ pin must be at a logic high level (V_{IHMC}).

It should be noted that a RESET generated by a WDT time-out does not drive the $\overline{MCLR}$ pin low.

6.11.2 WAKE-UP FROM SLEEP

The device can wake-up from SLEEP through one of the following events:

1. External RESET input on $\overline{MCLR}$ pin.
2. WDT wake-up (if WDT was enabled).
3. Interrupt from RB0/INT pin, RB port change, or data EEPROM write complete.

Peripherals cannot generate interrupts during SLEEP, since no on-chip Q clocks are present.

The first event ($\overline{MCLR}$ Reset) will cause a device RESET. The two latter events are considered a continuation of program execution. The $\overline{TO}$ and $\overline{PD}$ bits can be used to determine the cause of a device RESET. The $\overline{PD}$ bit, which is set on power-up, is cleared when SLEEP is invoked. The $\overline{TO}$ bit is cleared if a WDT time-out occurred (and caused wake-up).

While the SLEEP instruction is being executed, the next instruction (PC + 1) is pre-fetched. For the device to wake-up through an interrupt event, the corresponding interrupt enable bit must be set (enabled). Wake-up occurs regardless of the state of the GIE bit. If the GIE bit is clear (disabled), the device continues execution at the instruction after the SLEEP instruction. If the GIE bit is set (enabled), the device executes the instruction after the SLEEP instruction and then branches to the

SLEEP

Syntax:	[*label*] SLEEP
Operands:	None
Operation:	00h → WDT, 0 → WDT prescaler, 1 → $\overline{TO}$, 0 → $\overline{PD}$
Status Affected:	$\overline{TO}$, $\overline{PD}$
Description:	The power-down status bit, $\overline{PD}$ is cleared. Time-out status bit, $\overline{TO}$ is set. Watchdog Timer and its prescaler are cleared. The processor is put into SLEEP mode with the oscillator stopped.

Мікроконтролер PIC16F84 має лише базовий режим сну, однак у сучасніших мікроконтролерах реалізовано значно розвиненіші механізми зниженого енергоспоживання. Різні виробники пропонують власні набори режимів, які відрізняються глибиною відключення вузлів і швидкістю відновлення роботи.

Існують режими очікування, напівсну, глибокого сну та інші варіанти. Відмінності між ними визначаються насамперед роботою тактового генератора і частотою ядра, а також тим, які периферійні модулі залишаються активними. У легших режимах може знижуватися частота або вимикатися частина периферії, у глибоких – зупиняється майже вся система.

Деякі режими передбачають живлення лише критично важливих вузлів, наприклад модулів збереження даних або схем пробудження від зовнішньої події. Таким чином досягається мінімальне споживання при збереженні можливості відновити роботу без повного перезапуску. Різноманітність таких режимів дозволяє оптимізувати баланс між енергоспоживанням і функціональністю залежно від конкретного застосунку.

Режими зниженого енергоспоживання

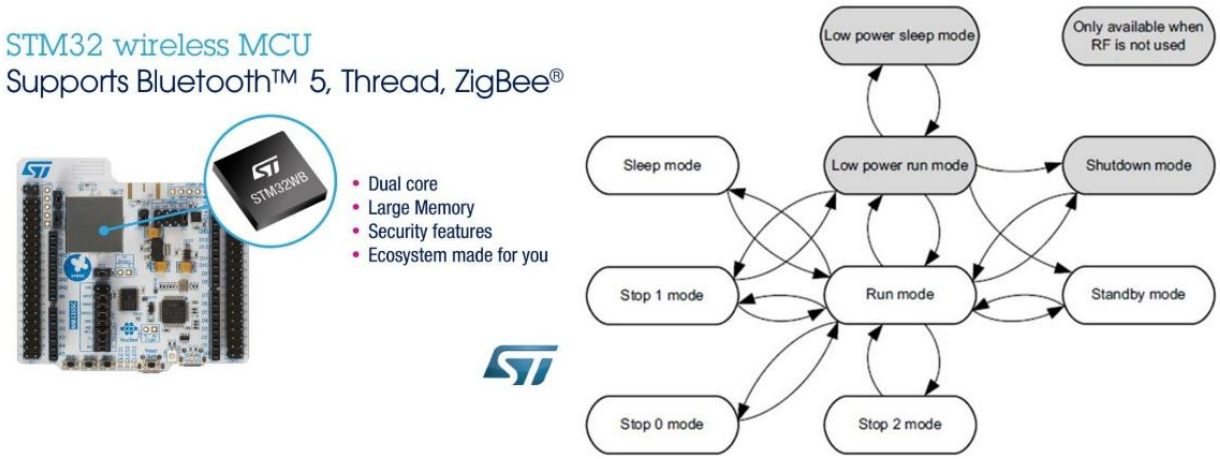
- **Режим очікування** (Idle Mode), при якому тактування ядра припиняється, але периферійні пристрої продовжують працювати на основній тактовій частоті
- **Режим напівсну** (Doze Mode), повністю аналогічний режиму очікування і відрізняється від нього тільки тим, що ядро тактується зниженою частотою
- **Режим сну** (Sleep Mode) – основний режим зниженого енергоспоживання, під час якого відключаються всі вузли, що не використовуються, у тому числі і ядро мікроконтролера
- **Режим глибокого сну** (Deep Sleep Mode) – режим максимально низького енергоспоживання, під час якого продовжують працювати лише окремі спеціально розроблені для цього режиму вузли
- **Низьковольтний режим сну** (Low-Voltage/Retention Sleep Mode) аналогічний режиму сну, але при цьому напруга живлення ядра і периферійних пристроїв знижується до мінімально можливого значення
- **Режим живлення від батареї** (VBAT Mode) – аварійний режим роботи, при якому основна напруга живлення зменшується до небезпечних значень (у цьому випадку живлення критично важливих вузлів здійснюється від окремого джерела живлення)

Як приклад сучасних рішень можна розглянути систему на кристалі компанії STM, у якій інтегровано не лише мікроконтролер, а й модуль Bluetooth. Такі SoC містять велику кількість режимів роботи, призначених для мінімізації енергоспоживання під час живлення від батареї.

Різноманіття режимів пов’язане з можливістю вибірково вимикати або обмежувати роботу окремих вузлів, змінювати частоту тактування, керувати живленням периферії та радіомодуля. Це дозволяє гнучко балансувати між продуктивністю та споживанням залежно від поточного стану системи.

Якщо живлення необмежене, наприклад від мережі, ці механізми можна практично не використовувати і працювати на постійній частоті. Однак для автономних пристроїв правильний вибір і керування режимами енергозбереження є критично важливими, оскільки саме вони визначають тривалість роботи від однієї батареї.

Система-на-кристалі STM32WB

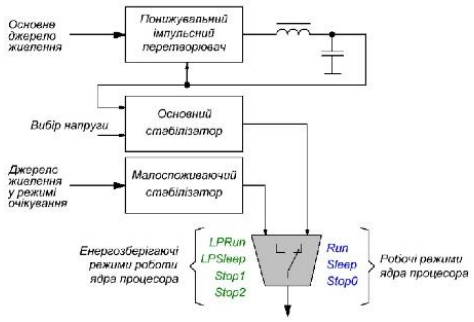


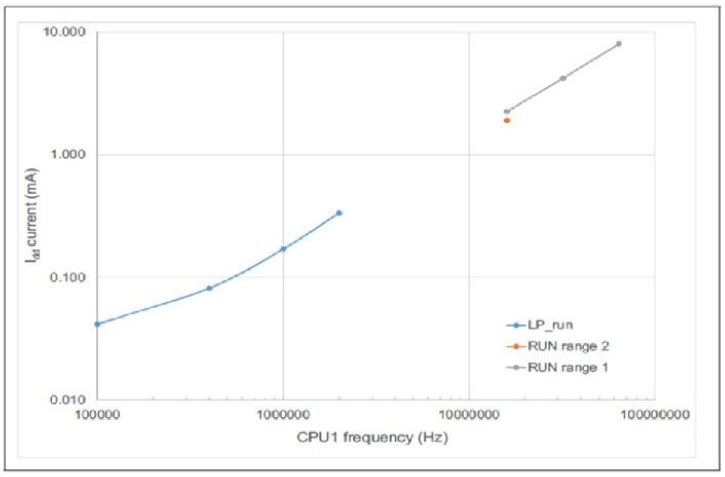
У сучасних системах на кристалі, окрім обчислювального ядра і радіомодулів, інтегруються також апаратні вузли керування живленням, які раніше не були характерні для мікроконтролерів. Це можуть бути лінійні стабілізатори напруги, імпульсні понижувальні перетворювачі та інші елементи системи живлення.

Такі вузли дозволяють гнучко керувати енергоспоживанням залежно від режиму роботи. Завдяки цьому одна й та сама система може змінювати споживаний струм на кілька порядків. У різних режимах різниця може досягати приблизно тисячократного значення, що є критично важливим для автономних пристроїв.

Подібний широкий діапазон регулювання дає змогу максимально ефективно використовувати енергію батареї, але вимагає від розробника розуміння принципів роботи цих режимів і правильного їх застосування в програмі. Без цього досягти тривалої автономної роботи практично неможливо.

Залежність споживаного струму системи-на-кристалі STM32WB від тактової частоти при різних режимах роботи



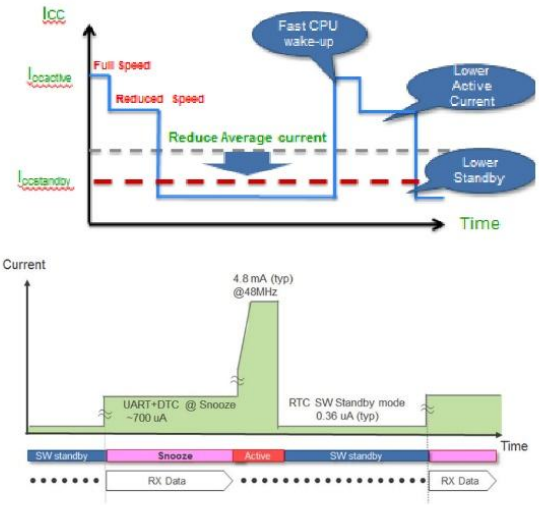


Основні методи зменшення енергоспоживання базуються на керуванні режимами роботи системи. Насамперед знижують тактову частоту ядра, по можливості зменшують напругу живлення, скорочують кількість виконуваних арифметико-логічних операцій. Останній фактор безпосередньо залежить від алгоритмів і якості програмного коду.

Додатково вимикають невикористовувані периферійні модулі та за можливості використовують спеціалізовані вузли з пониженим споживанням. Усе це покладається на розробника програмного забезпечення, який повинен правильно організувати роботу пристрою залежно від його функцій.

На відміну від криптографії або інтерфейсів зв'язку, універсальних готових рішень тут не існує, оскільки кожен застосунок має власний профіль роботи та навантаження. Тому оптимізація енергоспоживання виконується індивідуально і є складовою процесу розробки програмного забезпечення.

Методи зменшення енергоспоживання мікроконтролерів



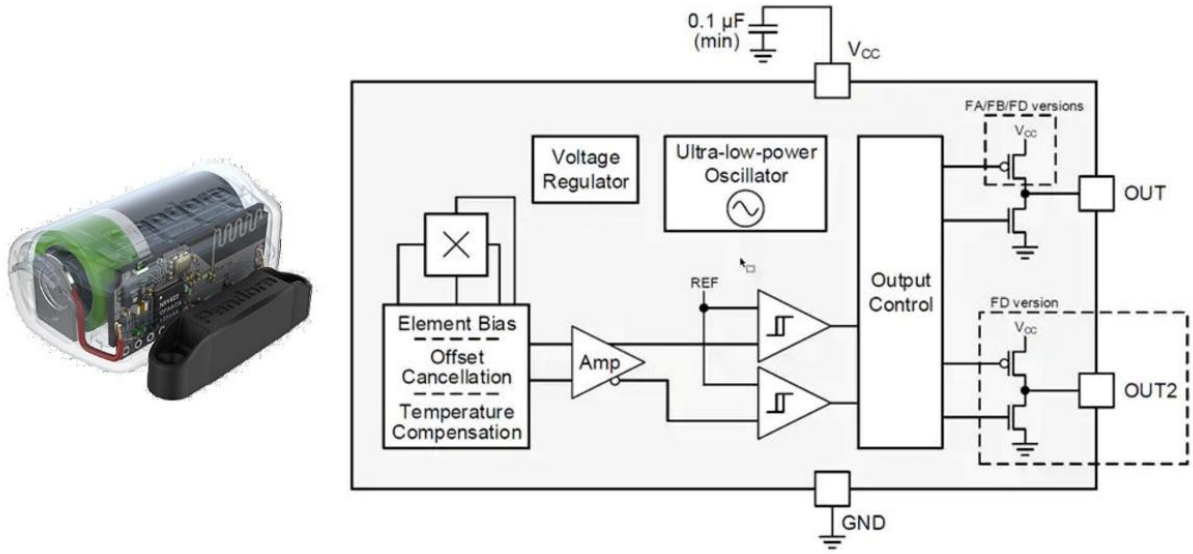
- Зменшення тактової частоти
- Зниження напруги живлення
- Зменшення кількості арифметико-логічних операцій
- Відключення модулів, що не використовуються у даний час
- Використання спеціалізованих вузлів, із зменшеним енергоспоживанням

Для повноцінного забезпечення низького енергоспоживання в пристроях інтернету речей використовують також спеціалізовані апаратні компоненти, спроєктовані для роботи в енергоощадних режимах. Прикладом є датчики на ефекті Холла, призначені для виявлення магнітного поля.

Такий датчик має цифровий вихід і підключається безпосередньо до входу мікроконтролера, повідомляючи лише факт наявності або відсутності магнітного поля, тобто фактично передає один біт інформації. Подібні рішення широко застосовуються, наприклад, у датчиках відкриття дверей або вікон.

Однак навіть такі прості сенсори можуть мати відносно високий струм споживання у безперервному режимі роботи. Якщо живити їх постійно від батареї, ресурс джерела живлення швидко вичерпається, що робить необхідним застосування додаткових методів керування живленням.

Структурна схема датчика Хола DRV5032 для пристроїв Інтернету-речей

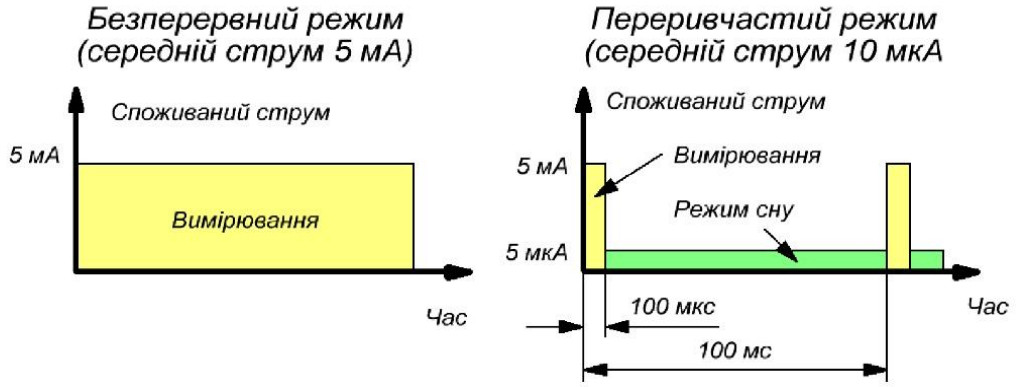


У таких сенсорах енергоефективність досягається апаратно. Усередині мікросхеми вбудовано власний таймер, який періодично переводить датчик у режим сну, залишаючи активним лише мінімально необхідний вузол контролю. Датчик не вимірює магнітне поле безперервно, а робить це через задані інтервали часу, після чого знову засинає.

Завдяки такому імпульсному режиму контролю середнє енергоспоживання значно зменшується. При цьому від розробника не потрібно жодних додаткових дій – достатньо правильно вибрати компонент. Існують повністю сумісні за виводами моделі з однаковою функціональністю, але з різницею у споживанні на порядки: наприклад, одна може споживати кілька міліампер, інша – десятки мікроампер, що дає сотні разів економії.

Такі спеціалізовані сенсори розроблені саме для автономних систем і дозволяють істотно продовжити термін роботи пристрою від батареї без ускладнення програмної частини.

Принцип роботи енергозберігаючого датчика Хола DRV5032



Використання переривчастого режиму роботи дозволило зменшити енергоспоживання у 500 разів

Створення пристрою інтернету речей потребує знань у багатьох галузях. Необхідно розуміти аналогову електроніку, програмування мікроконтролерів, питання інформаційної безпеки, організацію обміну даними та специфіку прикладної області.

Тому розробка таких систем зазвичай виконується командою фахівців різного профілю. Попри складність, початкових знань з мікроконтролерів достатньо, щоб розпочати роботу в цьому напрямку. Подальший розвиток залежить від практичного досвіду та самостійного поглиблення знань.

Висновки

При використанні готових рішень пристрої Інтернету-речей можна створити достатньо швидко

Навчальне видання

Русу Олександр Петрович

ПРОГРАМУВАННЯ МІКРОКОНТРОЛЕРІВ ТА РОБОТОТЕХНІКА В ЗАКЛАДАХ ОСВІТИ

Конспект лекцій