Міністерство освіти і науки, молоді та спорту України
Національний університет “Львівська політехніка”

Реєстр. №
від __.__.2012

Програмування мікроконтролерів систем автоматики

Конспект лекцій

для студентів базового напряму
050201 “Системна інженерія”

Затверджено
на засіданні кафедри
“Комп’ютеризовані
системи автоматики”
Протокол № 9 від 03.04.2012

Львів 2012

Програмування мікроконтролерів систем автоматики: конспект лекцій для студентів базового напряму 050201 “Системна інженерія” / Укл.: А.Г. Павельчак, В.В. Самотий, Ю.В. Яцук – Львів: Львівська політехніка. – 2012. – 143 с.

Укладач: А.Г. Павельчак, к.т.н., доцент bilyj@ukr.net
В.В. Самотий, д.т.н., професор
Ю.В. Яцук, к.т.н., ст. викладач

Відповідальний за випуск:
А.Й. Наконечний, д.т.н., професор

Рецензент: І.М. Бучма, д.т.н., професор

ЗМІСТ

I. ОСНОВНІ ПОНЯТТЯ………………………………………………… 3
1.1. Основні відмінності між МП та МК …………...……………...…. 3
1.2. Типи мікроконтролерів …..……………………………………... 6
1.3. Організація доступу до пам’яті ……………………………..….. 7
1.4. Системи команд – CISC та RISC ………………..………...….. 8
II. ЗНАЙОМСТВО З МК ATMEL AVR …………………..…………..… 9
2.1. Модель ATmega32A …………………………...…………...……. 9
2.2. Структура ядра AVR ……………………………………….……. 11
2.3. Організація пам’яті даних ……………………………………. 15
III. ПРОГРАМУВАННЯ AVR МОВОЮ АСЕМБЛЕР …..……..……… 17
3.1. Приклад простої програми для AVR ………..…………………. 17
3.2. Директиви та функції асемблера AVR ………………………… 19
3.3. Представлення чисел ………………………………………..…. 26
3.4. Структура асемблерної програми ……………………………. 30
3.5. Паралельні порти вводу/виводу ……………………...………. 31
3.6. Варіанти підключення кнопок та світлодіодів до МК ……...…. 37
3.7. Умовні та безумовні переходи та регістр стану SREG ……...…. 39
3.8. Використання стеку ……………………………...……………. 51
3.9. Підпрограми ……………………………………………..……. 52
3.10. Реалізація переривань …………………………………………. 54
Зовнішні переривання ……………………………………….. 55
3.11. Макроси ………………………………………………………… 59
3.12. Робота з даними у SRAM, FLASH та EEPROM ………...……. 62
SRAM …………………………………………...…………. 62
FLASH ………………………………………...…………. 65
EEPROM ……………………………………..………………. 68
3.13. Таймери ………………………………………………………. 71
3.14. Використання асинхронних таймерів ………………………. 79
3.15. ШІМ-модуляція ……………………………………………….. 86
3.16. Сторожовий таймер (WatchDog) ……………………………… 92
IV. ПРОГРАМУВАННЯ AVR МОВОЮ С …..…………………………. 94
4.1. Основні компілятори Сі для МК AVR ……………………..……. 94
4.2. Типи даних мови Сі компілятора WINAVR ……………………. 95
4.3. Бітова арифметика ………………………………………..……. 95
а. Бітові поля ……………………………………………………. 95
б. Порозрядні операції …………………………………………. 96
в. Встановлення чи очищення бітів ……...……………………. 98
г. Інверсія бітів ………………………………………………. 99

д. Перевірка значень бітів ………………………..…………… 100
4.4. Базова структура програми мовою Сі ………………..……… 101
4.5. Глобальні та локальні змінні, директива volatile ………………. 103
4.6. Робота з перериваннями …………………………………………. 105
4.7. Робота з даними в пам'яті програм ………………..……………. 106
4.8. Програмні затримки ……………………………………….…. 108
4.9. Організація передачі даних через UART/USART ……….……. 109
а. Формат передачі кадру даних UART ……………………….. 109
б. Підключення UART ……………………………………………. 110
в. Апаратна частина UART ……………………………………. 111
г. Швидкість прийому/передачі ……………….………………. 112
д. Передача та прийом даних, переривання модуля UART …. 114
4.10. Реалізація кільцевого буфера FIFO для UART ………….……. 117
4.11. Інтерфейс RS-232 …………………………………………….…. 121
4.12. Інтерфейс RS-485 ……………………………………………….. 123
4.13. Мультипроцесорний режим модуля UART ……………………. 128
4.14. Аналогово-цифровий перетворювач ..………………...………. 133
Перелік рекомендованої літератури ……………………………….. 143

I. ОСНОВНІ ПОНЯТТЯ

1.1. Основні відмінності між МП та МК.

Мікропроцесор (МП) – програмований пристрій, що приймає двійкові дані від пристрою вводу, обробляє ці дані відповідно до інструкцій, збережених у пам'яті, і видає результати на вихід. Іншими словами, МП виконує програму, збережену в пам'яті, та передає дані від та до зовнішнього світу через порти вводу/виводу. Загалом, МП називають серцем будь-якої мікропроцесорної системи, яке виконує усі операції, а також контролює решта системи.

Мікроконтролер (МК) – можна розглядати як спеціалізований комп'ютер-на-кристалі або одно-кристальний комп'ютер. Слово «мікро» натякає, що пристрій є маленький, а слово «контролер» – що пристрій може використовуватися для контролю однієї чи більше функцій об'єктів, процесів чи подій. МК також називають вбудо-ваними (embedded) контролерами, оскільки МК часто вбудовані у пристрої та системи, які вони контролюють.

МК містить спрощений процесор, пам'ять (RAM та ROM), порти вводу/виводу, периферійні пристрої такі, як лічильники/таймери, аналогово-цифрові перетворювачі і т.п., і все це інтегроване в один кристал. Ця особливість і відрізняє МК від мікропроцесорних систем. Натомість, МП має лише процесор із регістрами загального призначення, а периферійні пристрої розміщені зовні. Якщо мікропроцесорна система є системою загального призначення, що може бути запрограмованою для виконання великого числа різних функцій, то мікроконтролери призначені для однієї задачі і виконують лише одну конкретну програму. Ця програма збережена в ROM і, як правило, не змінюється.

На рис. 1.1 показана основна відмінність між мікропроцесор-ними системами та мікроконтролерами: мікропроцесорні системи для своєї функціональності потребують додаткових мікросхем, в той час як у МК функції усіх цих додаткових мікросхем є інтегровані у тому самому кристалі МК.

CPU (Central Processing Unit) – центральний процесор.

RAM (Random Access Memory) – пам'ять з довільним доступом, чи оперативний запам'ятовуючий пристрій. RAM призначена для

зберігання проміжних результатів та інших тимчасових даних протягом виконання програми.

ROM (Read Only Memory) – пам'ять тільки для читання, чи постійно запам'ятовуючий пристрій. ROM зберігає програмні інструкції (програму виконання) та таблиці з довідковими даними. У сучасних МК вона реалізована у вигляді flesh-пам'яті.

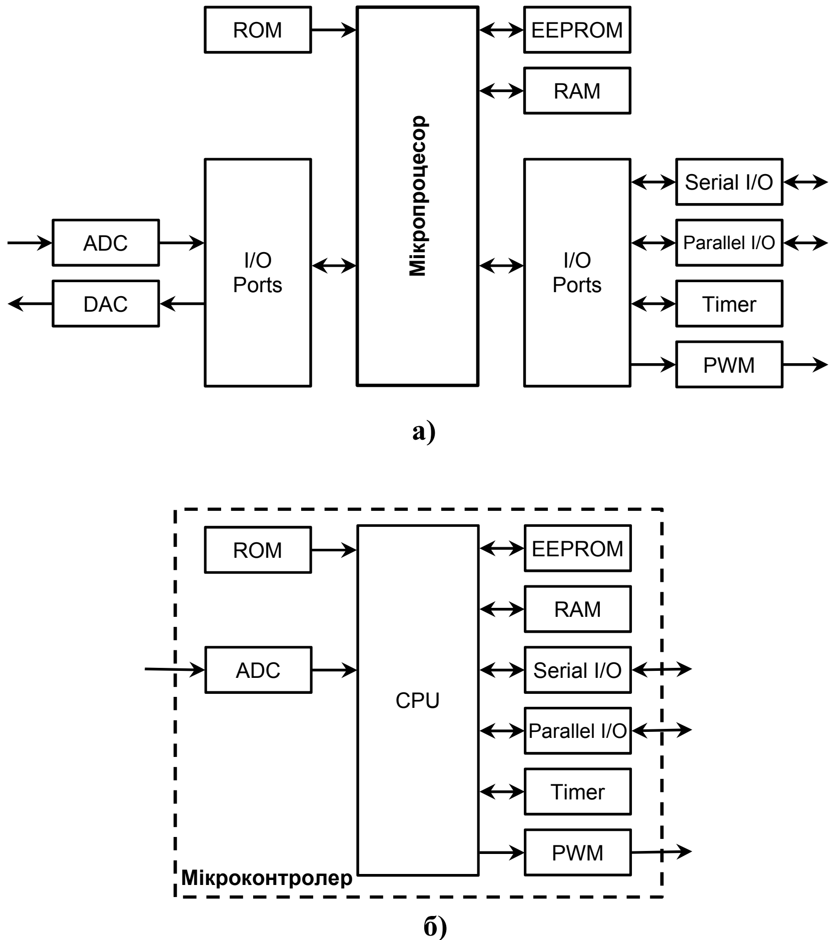


Рис. 1.1. Мікропроцесор у порівнянні з мікроконтролером:
а) конфігурація мікропроцесора; б) конфігурація мікроконтролера

EEPROM (Electrically Erasable and Programmable ROM) – енергонезалежна пам'ять даних. У сучасних МК вона представляє собою flash-пам'ять. Основна відмінність від flash-пам'яті програм (ROM) – це можливість вибіркового програмування окремих байтів, на відміну від поблокового програмування flash-пам'яті програм.

I/O Ports (input/output) – паралельні порти вводу виводу надають інтерфейс між МК та периферійними пристроями вводу/виводу, такими як: клавіатура, дисплей тощо. Ніжки портів можуть бути як двонаправленими, так і однонаправленими, або лише на вхід, або на вихід.

Serial I/O – послідовні порти для обміну даними, що реалізовують асинхронні (напр., UART) або синхронні (напр., SPI) інтерфейси обміну даними. Асинхронний інтерфейс використовує протокол зі стартовим та стоповим бітами для передачі та прийому. Стартовий та стоповий біти вбудовані у кожен байт даних. Синхронні інтерфейси використовують синхронізуючі імпульси для кожного біта.

Timer/Counter (таймер/лічильник) – використовують для відліку часу або/та меж часових інтервалів між подіями, підрахунку кількості подій та генерації швидкості передачі даних (у бодах) для послідовних портів. Таймери також можуть керувати певними I/O-ногами у МК, наприклад, здійснювати підрахунок кількості імпульсів, що поступають на його вхід, чи навпаки, виводити певні послідовності імпульсів.

PWM (Pulse Width Modulation, широтно-імпульсна модуляція ШІМ) – це спосіб кодування аналогового сигналу шляхом зміни ширини (тривалості) прямокутних імпульсів несучої частоти. Найбільш часто PWM використовують для керування моторами різних типів та активним навантаженням, наприклад, лампою розжарювання.

ADC (Analog to Digital Converter) – аналогово-цифровий перетворювач забезпечує інтерфейс для роботи з аналоговими пристроями, наприклад, з давачами, що видають аналогові електричні еквіваленти для фактичних фізичних параметрів, які ми хочемо контролювати.

DAC (Digital to Analog Converter) – цифро-аналоговий перетворювач забезпечує інтерфейс для роботи з виконавчими пристроями.

1.2. Типи мікроконтролерів.

– Вбудовані 8-розрядні МК.

  Мають просту систему команд та велику номенклатуру вбудованих пристроїв. Причиною життєздатності 8-розрядних МК є використання їх для керування реальними об'єктами, де застосовуються, в основному, алгоритми з переважанням логічних операцій, швидкість обробки яких практично не залежить від розрядності процесора.

  Приклади: 8051 чи MCS-51 (Intel); AVR (Atmel); PIC (Microchip); 68HC05, 68HC08, 68HC11 (Motorola); ST6, ST7, ST9, STM8L (STMicroelectronics); COP8 (National Semiconductor); TMS370 (Texas Instruments) та ін.

– 16-розрядні МК.

  Структури та системи команд зорієнтовані на прискорену реакцію на зовнішні події, і, відповідно, на таких МК можуть будуватися системи реального часу середньої продуктивності.

  Приклади: MCS-96 (Intel); PIC24 (Microchip); MSP430 (Texas Instruments); 68HC16 (Motorola); MB90 (Fujitsu); M16C (Mitsubishi) та ін.

– 32-розрядні МК.

  Велика частина з них побудовані на ядрі ARM та призначені для систем телефонії, оброблення та передачі інформації, телебачення.

  Приклади: STM32 (STMicroelectronics); Sitara ARM (Texas Instruments); HT32F125x (Holtek); LPC3000, LPC2900 (NXP); AVR32 (Atmel); PIC32 (Microchip) та ін.;

– Цифрові сигнальні процесори (DSP, Digital Signal Processor).

  Використовуються для складного математичного оброблення аналогових сигналів у режимі реального часу. Застосовуються у телефонії та зв'язку.

  Приклади: C5000, C6000 (Texas Instruments); ADSP-21xx, SigmaDSP (Analog Devices), MSC81xx (Freescale).

1.3. Організація доступу до пам'яті.

Існує дві фундаментальні архітектури, що використовуються процесорами для доступу до пам'яті: архітектура Фон-Неймана (Von Neumann architecture), відома ще як Прінстонська, та Гарвардська архітектура (Harvard architecture).

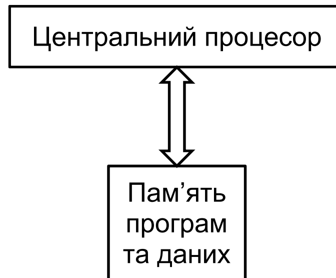


Рис. 1.2. Архітектура Фон-Неймана

Архітектура Фон-Неймана використовує одну пам'ять для зберігання як програмних інструкцій (команд), так і даних. В наявності є лише одна загальна шина для адрес та даних між процесором та пам'яттю (рис. 1.2). Команди та дані витягуються у послідовному порядку, таким чином обмежуючи швидкість передачі даних чи пропускну здатність.

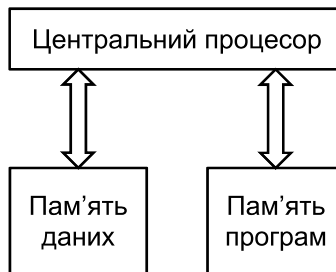


Рис. 1.3. Гарвардська архітектура

Гарвардська архітектура використовує фізично розділену пам'ять для програм та даних. Це, відповідно, вимагає окремих шин для програми та окремо для даних (рис. 1.3). У такій архітектурі команди та операнди можуть бути витягнуті одночасно з пам'яті, що

робить МК з такою архітектурою більш швидшими у порівнянні з МК, що використовують архітектуру Фон-Неймана. Також шина даних та шина програм можуть мати різну розрядність, що дає можливості для кращої оптимізації пам’яті даних та пам’яті програм у відповідності до архітектурних вимог.

Переваги архітектури Фон-Неймана полягають у спрощеній схемотехніці процесора та у гнучкості розподілу ресурсів між областями пам’яті, що дуже важливо в операційних системах реального часу. Архітектура Фон-Неймана стала основною архітектурою універсальних комп’ютерів, включаючи персональні.

Більшість сучасних МК використовують саме гарвардську архітектуру пам’яті.

1.4. Системи команд – CISC та RISC.

CISC (Complex Instruction Set Computer) — процесор з повним набором команд. Такі процесори мають у своєму авангарді велику кількість різноманітних команд, які в свою чергу можуть відрізнятися форматом та довжиною. Прості команди можуть бути виражені за допомогою короткого командного коду розміром в один байт, який виконується дуже швидко. Складні команди можуть складатися з декількох байт коду та займати багато часу на виконання. При цьому виконання певної як завгодно складної команди із системи команд процесора реалізовується апаратно усередині самого процесора. У систему команд CISC-процесора може входити, наприклад, обчислення квадратного кореня, що потребує багато десятків тактів. Додавання кожної нової команди призводить до збільшення загального числа транзисторів у процесорі. CISC-процесори також характеризуються великою кількістю методів адресації пам’яті та порівняно невеликою кількістю робочих регістрів. На практиці рідко використовується увесь перелік команд CISC-процесора, як правило не більше 20%.

RISC (Reduced Instruction Set Computer) — процесор зі скороченим набором команд. У процесорах з такою архітектурою використовується обмежений набір швидких команд з фіксованою шириною. Кожна команда повинна виконуватися, в ідеалі, за один машинний цикл, тому ви навряд чи знайдете в системі команд ділення. У таких процесорах міститься менше число транзисторів, що знижує їхню вартість та енергоспоживання. При цьому, як правило, підвищується їхня продуктивність. Але, там де CISC-процесор виконував одну

команду, для RISC-процесора необхідно писати невелику програму. RISC-процесори характеризується збільшеною кількістю регістрів загального призначення та простими способами адресації пам'яті.

Майже усі сучасні МК є побудовані за RISC-архітектурою. До МК із CISC-архітектурою відносяться МК фірми Intel з ядром MCS-51, які на сьогодні за ліцензією ще випускаються цілим рядом виробників. Істинними CISC-процесорами вважалися процесори ПК з архітектурою x86, але починаючи з Intel 486DX, вони є CISC-процесорами з RISC-ядром. Вони безпосередньо перед виконанням перетворюють CISC-інструкції x86-процесорів у більш простий набір внутрішніх інструкцій RISC.

II. ЗНАЙОМСТВО З МК ATMEL AVR

2.1. Модель ATmega32A.

Знайомство з архітектурою ядра AVR та навчання програмуванню МК цього сімейства протягом усього курсу лекцій будемо на прикладі моделі ATmega32A. Лабораторні макети, що будуть використовуватися на практичних заняттях, також побудовані на цій моделі.

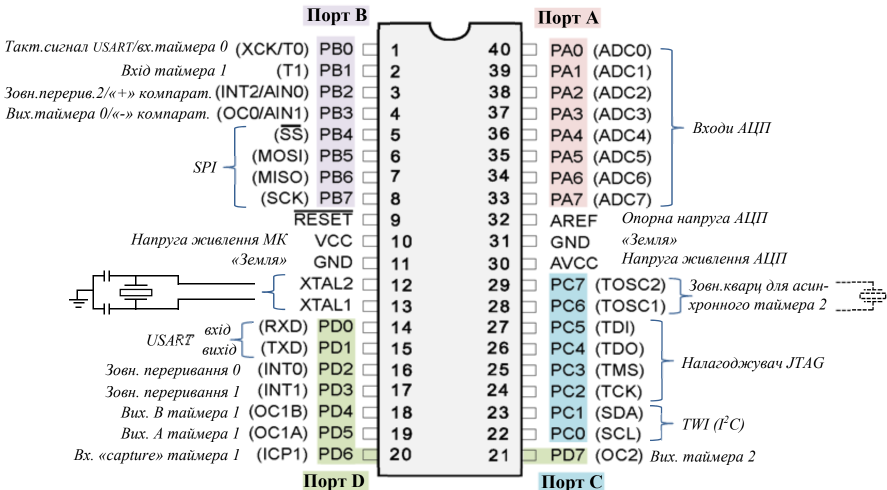


Рис. 2.1. Найменування виводів моделі ATmega32A у DIP-корпусі

Модель ATmega32A відноситься до платформи Atmel, сімейства Mega, має ядро AVR архітектури RISC гарвардського типу.

ATmega32A має в наявності 4 повних паралельних порти вводу/виводу: **Порт A** – виводи PA0, PA1, … , PA7 (ноги 40-33); **Порт B** – PB0, PB1, …, PB7 (ноги 1-8); **Порт C** – PC0, PC1, … , PC7 (ноги 22-29); **Порт D**: PD0, PD1, … , PD7 (ноги 14-21).

Окрім основного призначення, ноги цих портів також можуть виконувати інші функції, наприклад, ноги порту A також можуть служити як аналогові входи (ADC0, … , ADC7) для вбудованого АЦП, ноги PD0-PD1 (RxD, TxD) реалізують функції послідовного порту UART, ноги PC6-PC7 (TOSC1, TOSC2) дають можливість для підключення додаткового кварцу для асинхронного таймера МК тощо.

Особливості ATmega32A: 32 Кбайти флеш пам’яті програм, 2 Кбайти SRAM, 1 Кбайт EEPROM, два 8-бітні таймери (один з них асинхронний), один 16-бітний таймер, сторожовий таймер, 4 PWM канали, 8 каналів 10-бітного АЦП, послідовні інтерфейси (TWI (I^2C), USART, SPI), напруга живлення 2.7-5.5 В, частота тактування до 16 МГц.

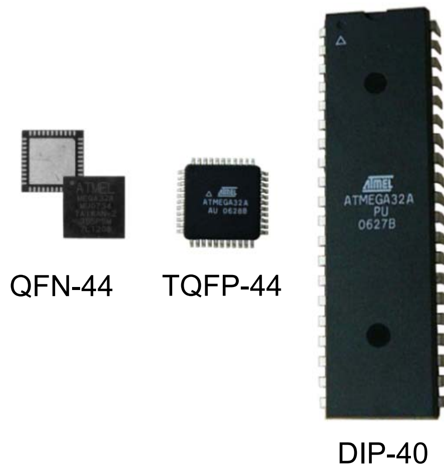


Рис. 2.2. Габарити та вигляд моделі ATmega32A у різних корпусах

Розшифрування: QFN (Quad-flat no-lead package);
TQFP (Thin Quad Flat Pack);
DIP (Dual Inline Package) .

2.2. Структура ядра AVR.

Основою будь-якого МК є обчислювальне ядро, яке також виконує і керування інтегрованою периферією. У всіх моделях AVR воно однакове, тому легко забезпечується переносимість коду в межах цілої лінійки.

Ядро AVR побудоване за **Гарвардською архітектурою**, згідно якої розділені не тільки види пам'яті (адресний простір пам'яті

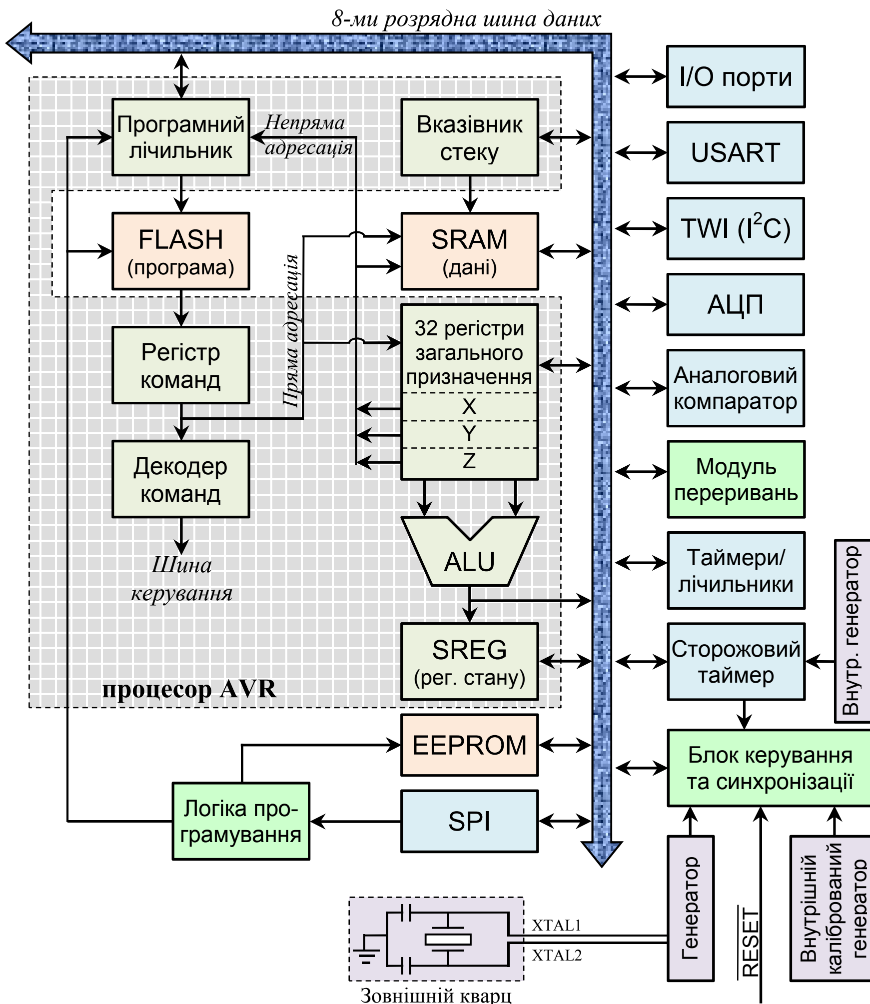


Рис. 2.3. Архітектура ядра AVR

програм та пам'яті даних), але і шини доступу до цих видів пам'яті. Кожна з областей пам'яті даних (SRAM та EEPROM) також розташовані у своєму власному адресному просторі. Це дає можливість зчитувати код програми паралельно з операціями читання/запису даних. Розділення шин доступу дозволяє використовувати для кожного типу пам'яті шини різної розрядності, при цьому способи адресації та доступу до кожного типу пам'яті також різні, та використовувати технологію конвеєризації, що підвищує загальну швидкодію. Конвеєризація полягає у тому, що під час виконання поточної команди здійснюється вибірка з пам'яті та дешифрування коду наступної команди.

Ядро AVR використовує вдосконалену (enhanced) **RISC** систему команд. Більшість арифметичних та логічних команд виконуються за 1 такт, але деякі команди (наприклад, команди виклику переривань та підпрограм, а також повернення з них) виконуються за 4-5 тактів. Розмір команди фіксований: 1 комірка пам'яті програм (16 біт) для переважної більшості команд та 2 комірки для команд, у яких один з операндів є 16-ти розрядною адресою.

При старті МК значення **програмного лічильника** (Program Counter) рівне $000, що є адресою першої команди в нашій пам'яті програм (FLASH). МК витягує з пам'яті програм два байти (код команди та її операнд) та віддає на виконання в **декодер команд**. Подальші дії вже залежать від самої команди. Якщо це проста команда для виконання певної роботи (арифметичної, логічної тощо), то ця робота буде виконана, а на наступному такті значення програмного лічильника буде збільшене, і з наступної комірки пам'яті будуть взяті чергові два байти команди та відправлені на виконання. Якщо ж зустрінеться команда переходу, тоді у програмний лічильник загрузиться адреса, що вказана у команді (абсолютний перехід), або лічильник збільшиться не на 1, а на необхідну кількість (відносний перехід), та на наступному такті МК візьме команду вже з нової адреси.

Декодер команд (Instruction Decoder) отриману команду віддає логіці блоку керування, який, у свою чергу, заставляє решта блоків виконувати потрібні дії у потрібному порядку.

ALU – Arithmetic Logic Unit (Арифметично-логічний пристрій) виконує усі арифметичні, логічні та частину бітових операцій. На вхід ALU можуть поступати лише **регістри загального призначення** (R0-R31) та константи. Операції можуть здійснюватися над одним або двома регістрами загального призначення, або регістром та константтою у другому операнді. Якщо команди працюють з константами, тоді у якості першого операнда можуть використовуватися лише регістри із

другої половини (R16-R31) регістрів загального призначення. Команди 16-розрядного додавання та віднімання, наприклад, працюють лише з чотирма останніми парами регістрів. Регістри загального призначення розміщені в адресному просторі SRAM (пам'яті даних), але для швидкої роботи з ними винесені фізично за межі SRAM.

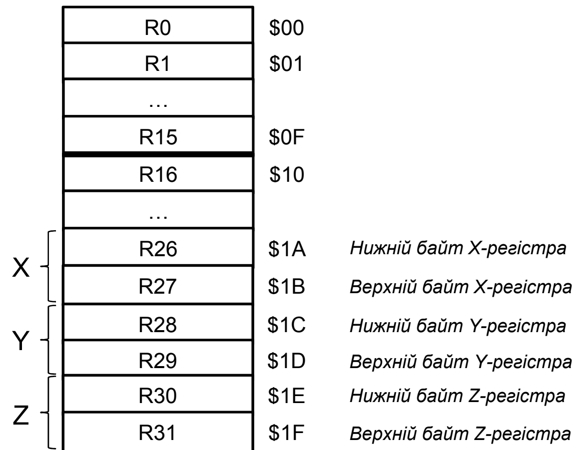
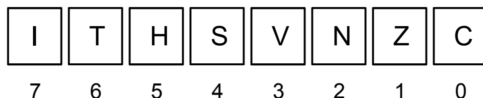


Рис. 2.4. Регістровий файл

Регістри загального призначення можна поділити на 3 групи:

- Молодші R0 … R15. Ці регістри не можуть працювати з командами, що оперують з константами, наприклад, ми не можемо в R0 записати число (константу), але можемо скопіювати в нього число з будь-якого іншого регістра.
- Старші R16 … R31. Повноцінні регістри, що працюють майже зі всіма командами.
- Індексні R26 …R31. Вони можуть використовуватися як звичайні регістри загального призначення, але крім цього можуть утворювати регістрові 16-розрядні пари X(R26:R27), Y(R28:R29), Z(R30:R31), що використовуються як вказівники для непрямої адресації пам'яті.

Під час виконання арифметичних, логічних та порозрядних операцій ALU формує певні інформативні ознаки результату виконання у вигляді бітів регістру стану **SREG** (Status Register), які потім можуть бути використані в програмі для подальших арифметично-логічних операцій чи команд умовних переходів. Такі інформативні біти часто називають прапорцями.



C – прапорець переносу.
Z – прапорець нуля.
N – прапорець від'ємного результату.
V – прапорець переповнення у двійковій арифметиці.
S – прапорець знаку.
H – прапорець часткового переносу.
T – користувацький прапорець.
I – прапорець дозволу глобальних переривань.

Про ці прапорці буде детально розказано при розгляді умовних операцій.

Для збереження тимчасових даних та адрес повернення після виконання переривань чи підпрограм використовується стек. Стек розміщується в SRAM, як правило, у її кінцевій частині, та при заповненні росте у напрямку початку оперативної пам'яті. **Вказівник стеку** вказує на його початок, та представляє собою два 8-бітних регістра SPH:LPL. При використанні підпрограм та переривань вказівник стеку обов'язково має бути проініціалізований.

Модуль переривань має свої керуючі регістри з додатковим прапорцем дозволу глобального переривання у регістрі стану SREG. Усі переривання мають свої вектори переривань у таблиці векторів переривань, яка розташована на початку пам'яті програм (FLASH). Порядок пріоритетів переривань відповідає місцю знаходження вектора переривань у таблиці – переривання з найменшою адресою вектора має найвищий пріоритет.

Пам'ять програм (FLASH) у МК AVR містить команди для керування процесором, а також може використовуватися для зберігання константних табличних даних. FLASH має 16-бітну організацію пам'яті та її розмір сягає від 1 до 256 КБайт, у залежності від моделі. FLASH пам'ять допускає до 10 тис. разів перезапису.

EEPROM (енергонезалежна пам'ять даних) використовується для тривалого зберігання усяких налаштувань, переналаштувань, зібраних даних і т.п., тобто всіх даних, які слід зберегти до наступного запуску МК. Кількість циклів перезапису сягає 100 тис., що є небагато у порівнянні зі SRAM. Читання даних триває близько 4 тактів, однак запис відбувається дуже довго – близько 2-9 мсек. (це час виконання декількох тисяч команд).

2.3. Організація пам'яті даних.

Адресація пам'яті даних здійснюється побайтно та повністю лінійна, без будь-яких поділів на сторінки, сегменти чи банки. Регістри заг. призначення, регістри керування та периферії, внутрішня оперативна пам'ять знаходяться в одному адресному просторі (рис. 2.5).

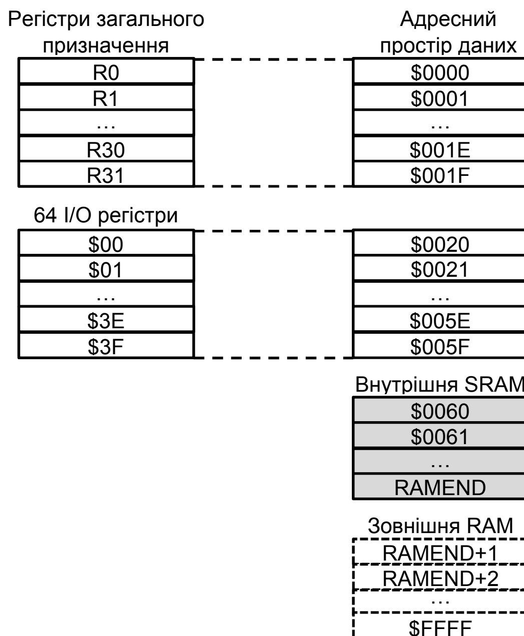


Рис. 2.5. Адресний простір пам'яті даних МК AVR

Адресний простір SRAM поділений на декілька областей:

- $0000-$001F – знаходяться 32 регістри загального призначення. В асемблері ми звертаємося до них за їхніми безпосередніми назвами R0, R1 і т.д., однак можемо зчитувати та записувати у них дані за їхніми повними адресами у SRAM за допомогою групи команд LOAD/STORE.
- $0020-$005F – знаходяться 64 регістри вводу/виводу (порти, таймери, АЦП, регістри керування і т.п.). Ці регістри також мають свою внутрішню адресацію від $00 до $3F, яка використовується разом зі швидкими командами IN/OUT. Але можемо також працювати з ними через команди LOAD/STORE, використовуючи їхні повні адреси у SRAM.

- $0060-RAMEND – безпосередньо розміщується внутрішня статична оперативна пам'ять даних. Значення RAMEND залежить від моделі МК. Розміщувати наші дані в пам'яті можемо лише починаючи з адреси $0060.

Слід зазначити, що регістри загального призначення та регістри вводу/виводу не віднімають простір у пам'яті даних, а лише відсувають початок її адресації. Наприклад, модель ATmega32A має на борту 2 КБайти SRAM (2048 байт чи 0x800). Відповідно, її значення RAMEND = 0x85F= 0x1F(рег.заг.призн.) + 0x3F(I/O порти)+ 0x800 (SRAM).

У деяких «нафаршированих» периферією моделях може не вистачати виділеної області адресації для регістрів вводу/виводу. Тому для таких моделей одразу після області регістрів вводу/виводу зарезервована додаткова область пам'яті 0x0060 - 0x00FF для додаткових 160 регістрів вводу/виводу. Слід зазначити, що з ними не працюють швидкі команди IN/OUT, і єдиний спосіб роботи з ними через команди LOAD/STORE. Відповідно, початок внутрішньої пам'яті даних вже починається з адреси 0x0100.

Ще один важливий момент стосується команд встановлення/очищення окремих бітів регістрів вводу/виводу SBI/CBI та команд перевірки стану біта у регістрі вводу/виводу SBIS/SBIC. Вони працюють лише з молодшими 32 регістрами вводу/виводу, що мають внутрішню адресу $00-$1F.

Встановлення додаткової зовнішньої пам'яті RAM передбачено лише на окремих моделях, наприклад ATmega162.

Зауваження: *опис усієї інтегрованої периферії AVR та програмування її функціонування буде детально описано по частинах протягом усього курсу.*

III. ПРОГРАМУВАННЯ AVR МОВОЮ АСЕМБЛЕР

3.1. Приклад простої програми для AVR.

Знайомство з технікою програмування МК AVR мовою асемблер почнемо з розгляду структури простої програми. Усі приклади програм у конспекті лекцій наведені для програмного середовища AVR Studio.

```
.include "m32Adef.inc"      ; підключення бібліотечного файлу МК

        .DSEG               ; SRAM – сегмент даних (оперативна пам'ять)

        .CSEG               ; FLASH – сегмент програмного коду
        .org    $000   ┐
        jmp     reset  │
        ; .......      ├ вектор переривань
        .org    $028   │
        reti           ┘
reset:
        ;ініціалізація периферії
        ldi     r16, Low(RAMEND)
        out     SPL, r16
        ldi     r16, High(RAMEND)
        out     SPH, r16

        ldi     r16, 0x00       } Присвоєння регістрам
        ldi     r17, 0xFF       } загального призначення значень

        out     PORTA, r16      } Порт А -- працює на вихід
        out     DDRA, r17

        out     PORTB, r17      } Порт В -- працює на вхід
        out     DDRB, r16       } з підтягуючими резисторами

main:
        ОСНОВНИЙ
        ПРОГРАМНИЙ ЦИКЛ
        rjmp    main

        ПІДПРОГРАМИ

        .ESEG   ;EEPROM – сегмент для енергонезалежних даних
```

Оскільки кожна модель МК AVR має на борту певний визначений для неї набір периферії, то відповідно адреси регістрів, що відповідають за визначені пристрої периферії, відрізняються у кожної моделі. І тому, щоб не мати справи з адресами регістрів, а лише з їхніми іменами (визначеними у середовищі AVR Studio), усі відповідності між іменами та адресами регістрів винесені в окремі бібліотечні файли. Першим ділом при написані програми мовою асемблер у середовищі AVR Studio ми повинні підключити за допомогою директиви .include необхідний бібліотечний файл для вибраної моделі МК. У нашому випадку для моделі ATmega32A бібліотечний файл має назву "m32Adef.inc".

Компілятор AVR Studio має певний набір директив, які допомагають писати прості та ефективні програми на асемблері. Директиви .DSEG, .CSEG. та .ESEG умовно розбивають код програми на сегменти, в яких будуть міститися відповідні дані – для SRAM, для FLASH і для EEPROM. Директива .org вказує, що у цьому місці програми встановлюється конкретне значення адреси у пам’яті програми.

У наведеному прикладі програмний код (після директиви .CSEG) починається записом вектора визначених переривань. Перше переривання, скид МК, здійснює стрибок на програмну мітку reset, з якої починається ініціалізація наявної периферії та закінчується основним програмним циклом – програмною петлею на мітку main.

Навіщо зациклювати основну програму? Якщо ж немає в кінці написаного коду програми переходу на початок мітки main, тоді програма буде виконуватися далі, до кінця FLASH (а ATmega32A має аж 32 кБайти). У пустій області програмної пам’яті прописані значення FF, і відповідно МК пройдеться по них до кінця об’єму FLASH, ніби виконуючи пустий оператор, а потім продовжиться виконання з початку програми, тобто стрибок на мітку reset, ініціалізація периферії і т.п.

Оскільки наша робоча програма є зациклена, то одразу після команди переходу на мітку main можемо розміщувати необхідні підпрограми, до яких ми будемо звертатися з основного програмного циклу.

Введення чи зчитування значень з регістрів периферії ми можемо здійснювати лише через посередництво регістрів загального призначення R0-R31. Тобто, для того щоб записати значення у якийсь з периферійних регістрів, ми спершу присвоюємо це значення якомусь з регістрів загального призначення, а потім з цього регістра пересилаємо в периферійний регістр (рис. 3.1).

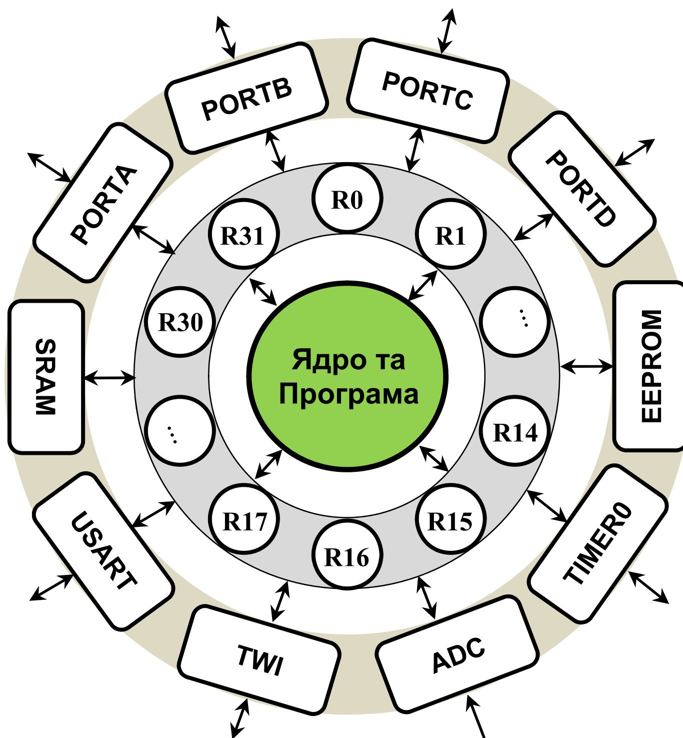


Рис. 3.1. Регістри загального призначення у ролі посередників

3.2. Директиви та функції асемблера AVR.

Асемблер МК AVR підтримує певне число директив. Ці директиви не транслюються безпосередньо в оперативний код програми. Натомість, вони використовуються для розмітки пам'яті, визначення макросів, ініціалізації пам'яті тощо. Вони є інструментом в руках програміста та спрощують йому життя. Розглянемо почергово основні директиви асемблера AVR.

.include – ця директива вказує компілятору зчитати вказаний файл та включити його вміст у поточну програму. Вміст зчитаного файлу вставляється у точку виклику цієї директиви. Включені файли також можуть містити .include директиви.

.exit – ця директива вказує компілятору на те, що досягнуто кінця файлу, і код, що розміщений після цієї директиви, ігнорується. Наприклад, ми включаємо за допомогою директиви .include певний файл з програмним кодом у наш проект і хочемо з метою уникнення

конфліктів імен, щоб включило не весь його вміст, а лише першу частину . Для цього у визначеному місці вставляємо директиву .exit, і тоді весь решта код після цієї директиви не буде включатися у наш проект.

У МК є в наявності три види пам’яті: пам’ять програм (FLASH), оперативна пам’ять (SRAM) та енергонезалежна пам'ять даних (EEPROM), і відповідно, в програмі передбачена можливість розмітки цих областей та при необхідності – ініціалізації значеннями.

```
.include "m32Adef.inc"

;============== SRAM – сегмент даних ==============================
          .DSEG

;============== FLASH – сегмент програмного коду ==================
          .CSEG

;============== EEPROM – сегмент для енергонезалежних даних ======
          .ESEG
```

Це свого роду шаблон для будь-якого нового проекту.

.DSEG (Data segment) – визначає початок сегменту даних. У цьому сегменті ми можемо лише здійснювати розмітку області SRAM за допомогою директиви .byte (та міток). Іншими словами, ми резервуємо місце для наших даних в оперативній пам’яті, помічаючи ці зарезервовані області мітками, адреси яких визначає компілятор.

.CSEG (Code segment) – визначає початок сегменту, де розміщується наша основна виконавча програма.

.ESEG (EEPROM segment) – визначає початок сегменту, що відноситься до енергонезалежних даних. У цій області ми можемо виконувати розмітку пам’яті як для розміщення конкретних даних, так і резервування простору для програмного збереження наших даних.

.byte – резервує вказану кількість байтів пам’яті в областях SRAM та EEPROM. Не може використовуватися у сегменті програмного коду.

```
          .DSEG
value:    .byte     1              ;резервує 1 байт для мітки value
vector:   .byte     3              ;резервує 3 байти для мітки vector
table:    .byte     10             ;резервує 10 байтів для мітки table
```

Значення адрес наших міток визначає компілятор. Якщо у моделі ATmega32A значення SRAM починається зі значення 0x060, то для цих міток компілятор визначить такі адреси:

```
value    = 0x060
vector   = 0x061
table    = 0x064
```

За замовчуванням значення пам'яті даних мають значення FF.

Для збереження табличних константних значень, тобто таких, які не змінюються в процесі виконання програми, наприклад таблиця значень синусів та косинусів, може використовувати як пам'ять EEPROM, так і пам'ять програм FLASH. Розміщення масивів даних в цих сегментах здійснюється за допомогою таких директив:

.db (define constant byte(s)) – вказує на розміщення масиву байтів. Кожне значення (вираз) масиву може приймати значення від -128 до 255. Якщо значення виражене від'ємним числом, тоді воно буде розміщене у пам'яті програм чи EEPROM як 8-ми бітне число у доповнюючому двійковому коді. Масив повинен містити щонайменше одне значення, два і більше значень відокремлюються між собою комами. Якщо директива .db вказана у сегменті .CSEG та масив містить більше ніж одне значення, тоді значення пакуються в пам'яті програм так, що два байти розміщуються в кожному слові пам'яті програм. Якщо масив містить непарне число значень, тоді останнє значення буде розміщене у своєму власному слові програмної пам'яті, а невикористана половинка програмного слова буде встановлена у нуль.

.dw (define constant word(s)) – вказує на розміщення масиву слів (2 байтних значень). Кожне значення масиву може приймати значення від -32768 до 65535. Якщо значення виражене від'ємним числом, тоді воно буде розміщене у пам'яті програм чи EEPROM як 16-ми бітне число у доповнюючому двійковому коді.

```
        .CSEG
v1:     .db     0, 255, 0b01010101, -125, 0xb3
v2:     .db      "Hello my dear friend"
v3:     .dw     -30125, 0xa0b5, 0b1010101010101010, 56, 65535

        .ESEG
V4:     .db     1, 2, 3
```

.dd (define constant doubleword(s)) та .dq (define constant quadword(s)) – ці директиви подібні до попередніх та використовуються для представлення масивів з 32 та 64-бітних значень.

.org (set program origin) – ця директива встановлює абсолютне значення адреси для комірки пам’яті. Може використовуватися у всіх трьох сегментах пам’яті: даних, програми та EEPROM. Є певні особливості щодо його використання. Якщо директива розміщується у сегментах SRAM та EEPROM, то необхідно враховувати, що адресація у них побайтна, а якщо розміщується у сегменті програми, тоді слід пам’ятати, що тут адресація послівна (по 2 байта). Ще один момент стосується пам’яті даних SRAM. Якщо вказати цю директиву з параметром 0, тоді ми будемо адресувати область, в якій розміщені регістри загального призначення. Тому для області даних розмітку слід робити, починаючи зі значення 0x60 чи навіть зі 0x100 (для МК з розширеною областю пам’яті для регістрів вводу/виводу). У прикладі простої програми ми зустрічали ці директиви при записі вектора переривань. Вони там необхідні, оскільки кожне переривання здійнює перехід на чітко встановлену адресу в межах цього вектора, що розміщений на початку області програмного коду.

```
        .DSEG
        .org    $150    ;встановлює адресу SRAM у значення 0x150
var:    .byte   1       ;резервує 1 байт у SRAM за адресою 0x150

        .CSEG
        .org    $40     ;встановлює Програмний лічильник у значення 0x40
        inc     r0      ;виконується якась робота
```

.def – ця директива дозволяє встановити символьні псевдоніми для регістрів загального призначення. В процесі написання програми ми для роботи з нашими змінними вибираємо певні регістри загального призначення. Наприклад, r0 – секунди годинника, r1 – хвилини, r2 – години, r16 – тимчасові проміжні значення і т.п. В процесі написання програми ми можемо заплутатися в тому, які регістри вже використані, а які ще вільні. Якщо ж захочемо за якоюсь змінною величиною закріпити інший регістр загального призначення, тоді це теж викличе проблеми, оскільки потрібно буде виловити усі розміщення попередньо-вибраного нами регістра. Тому правильним підходом є на початку програми визначати таблицю відповідних символьних псевдонімів для наших регістрів загального призначення, і у програмі вже використовувати їх.

```
.def    _second  = r0
.def    _minute  = r1
.def    _hour    = r2
.def    _temp    = r16
.def    _temp2   = r17

        .CSEG

        ldi     _temp, 25          ;встановлює r16 у значення 25
        mov     _minute, _temp     ;пересилає значення з  r16 у r1
```

Для одного регістра загального призначення може бути одночасно визначено декілька псевдонімів. Про це звісно компілятор видасть повідомлення. Такий підхід буває зручним для розділення інформативних термінів. При цьому звертатися до вибраного регістра не має значення за яким псевдонімом.

```
.def    _bin     = r10
.def    _low     = r10
```

.equ – ця директива закріплює за символьними мітками константні числові значення чи вирази, які в процесі компіляції будуть підмінені у програмі замість міток. Наприклад, у нас в програмі часто використовується певна константа, і скажімо, в певний момент з'ясується, що ми її невірно вибрали. Доведеться дуже ретельно виловлювати усі її появи у програмі, щоб випадково не пропустити. Натомість, оголосивши на початку програми за допомогою директиви .equ мітку, проблема знімається сама собою.

```
.equ    XTAL = 8000000
.equ    BaudRate = 9600
.equ    BaudDiv = XTAL / (16 * BaudRate) - 1
```

.set – за своєю роботою ця директива, як і .equ, закріплює за міткою числове значення, але на відміну від попередньої, вона дозволяє перевстановлювати по ходу програми значення для цієї мітки.

```
.set    Foo = 5
loop:
        ldi     r16, Foo           ; r16 = Foo = 5 (!)
.set    Foo = 10
        ldi     r16, Foo           ; r16 = Foo = 10
.set    Foo = Foo + 10
        ldi     r16, Foo           ; r16 = Foo =20
        rjmp    loop
```

У наведеному прикладі директива .set розбиває програму на три підсегменти, і в кожному з них мітка Foo має своє визначене значення. Навіть у циклі, де Foo приймає нове значення, при переході на початок циклу там буде діяти інше значення для Foo, яке визначене ще перед початком циклу. Тому такі маніпуляції з директивою .set слід застосовувати у програмі з обережністю.

Для створення макросів передбачені директиви початку .macro та кінця макросу .endmacro або .endm. На відміну від виклику підпрограм, де здійснюється перехід за міткою та повернення у вихідну точку, виклик макроса означає, що компілятор виконає вставку коду макроса у точку виклику. Скільки раз макрос буде викликаний, на стільки ж і збільшиться об’єм вихідного програмного коду.

```
.macro  Addition
        clr     @0
        add     @0, @1
        add     @0, @2
.endmacro

        .CSEG
        ldi             r18, 5     ; r18 = 5
        ldi             r19, 8     ; r19 = 8
        Addition  r10, r18, r19    ; r10 = r18+r19 = 13
```

Макрос може приймати до 10 параметрів. Посилання на ці параметри позначаються в середині макроса як @0-@9. Порядок слідування визначається при виклику макроса. У наведеному прикладі @0 це r10, @1 – r18, а @2 – r19. Під час компіляції при підстановці коду макроса у точки виклику замість параметризованих змінних підставляються параметри, що вказані через кому після імені викликаного макроса.

Асемблер також підтримує умовну компіляцію, і для неї використовуються такі директиви: .else, .elif, .endif, .error, .if, .ifdef, .ifndef, .message. Це дає можливість писати програми одразу для цілого ряду моделей МК.

Компілятор AVR Studio має у своєму арсеналі окрім директив ще також і набір функцій, які обчислюються в процесі компіляції (табл. 3.1). Ці функції мають суто допоміжний характер та застосовуються для спрощення обчислень необхідних величин, що використовуються як константи для нашої основної програми. Вони не мають відношення до математичних бібліотек, які використовуються для обчислень при роботі МК.

Таблиця 3.1. Основні функції компілятора AVR Studio

Low(*вираз*)	Повертає молодший байт від числового виразу
High(*вираз*)	Повертає другий байт від числового виразу
Byte2(*вираз*)	Аналогічна функції high()
Byte3(*вираз*)	Повертає третій байт від числового виразу
Byte4(*вираз*)	Повертає четвертий байт від числового виразу
Lwrd(*вираз*)	Повертає 0-15 біти від числового виразу
Hwrd(*вираз*)	Повертає 16-31 біти від числового виразу
Page(*вираз*)	Повертає 16-21 біти від числового виразу
Exp2(*вираз*)	Повертає значення 2 до степені виразу
Log2(*вираз*)	Повертає цілу частину від двійкового логарифма

```
.equ    foo   =0x1234
.equ    foo2 =0xABCDEF89

        .CSEG
        ldi     r16, Low(foo)           ; r16 = 0x34
        ldi     r17, High(foo)          ; r17 = 0x12
        ldi     r18, Low(foo2)          ; r18 = 0x89
        ldi     r19, Byte2(foo2)        ; r19 = 0xEF
        ldi     r20, Byte3(foo2)        ; r20 = 0xCD
        ldi     r21, Byte4(foo2)        ; r21 = 0xAB
        ldi     r22, Exp2(3)            ; r22 = 8
        ldi     r23, Log2(17)           ; r23 = 4

table:  .dw     Lwrd(foo2), Hwrd(foo2)  ; 0xEF89, 0xABCD
```

Також компілятор AVR Studio має ряд арифметичних, порозрядних, логічних та умовних операцій. Вони, як і функції, мають допоміжний характер і спрощують обчислення необхідних величин, та обробляються на етапі компіляції.

Таблиця 3.2. Перелік операцій компілятора AVR Studio

Арифметичні	+ - * /
Порозрядні	~ & | ^ << >>
Логічні	! && ||
Умовні	< > <= >= == != ?

Наведені операції за своєю дією ідентичні аналогічним операціям мови C++, за винятком одного моменту, що стосується значення результату логічних та умовних операцій. Воно є числом 1 або 0, у залежності від результату операції. Наприклад, логічне && повертає 1, якщо обидва значення є більшими за нуль, та повертає 0, якщо

значення рівні 0. Оператор рівності == повертає 1, якщо порівнювані значення рівні, інакше поверне число 0.

```
ldi        r16, 25*(c1==c2) + 1       ;якщо c1==c2, тоді  r16 = 26
                                      ; інакше r16 = 1
```

Коментарі в асемблері можуть записуватися як в класичному виді за допомогою ‘;’, так і в Сі-стилі.

```
; класичний асемблер ний коментар, діє до кінця стрічки
// Сішний рядковий коментар, діє до кінця стрічки
/* Сішний блоковий коментар,
 обгороджений колючками текст
компілятором ігнорується */
```

Довжина рядка в асемблерному коді обмежена 120 символами.

3.3. Представлення чисел.

Основним типом чисел в асемблері є цілочисельний 8-бітний тип. Асемблер підтримує різні представлення чисел:

- Десяткове (за замовчуванням): 26, 255;
- Шістнадцяткове (С та Pascal нотації): 0x1A, $FF, $1A, 0xFF;
- Двійкове: 0b00011010, 0b11111111;
- Вісімкове (нуль спереду): 032, 0377.

Асемблер може працювати як з беззнаковими цілими, так і зі знаковими числами. Для роботи з від’ємними числами використовується їхнє представлення у доповнюючому коді. Для представлення 8-бітного від’ємного числа від значення 256 віднімається це число без знаку, наприклад, -10 у доповнюючому 8-бітному числі це 256-10=246. Для асемблера значення -10 та 246 є ідентичними, тому аналіз значень покладається суто на програміста. Використання доповнюючого коду є природним для обчислювальної техніки, бо при цьому спрощується сам механізм арифметичних операцій з числами. У таблиці 3.3 наведено порівняльне співвідношення між знаковими значеннями та їхніми представленнями у 8-бітному форматі в регістрах.

Таблиця 3.3. Представлення 8-бітних знакових чисел

-4	-3	-2	-1	0	1	2	3
252	253	254	255	0	1	2	3
11111100	11111101	11111110	11111111	00000000	00000001	00000010	00000011

Ознакою від'ємного числа у двійковому виді є наявність «1» у старшому розряді. Про отриманий від'ємний результат сигналізує відповідний прапорець регістру стану SREG. І знову ж таки, МК не відрізняє знакових чисел від беззнакових, тому контроль покладається на програміста.

У МК AVR також підтримується апаратне множення чисел у дробовому форматі 1.7. Дамо пояснення про формати такого типу.

Формат «n.q» позначає дробове число з n розрядами (двійковими цифрами) зліва від десяткової крапки та q розрядами справа від неї. Дробові числа для 8-бітного МК AVR представляються у форматі 1.7, тобто 1 розряд під цілу частину чи знак та 7 розрядів під дробову частину.

Для беззнакового дробового числа у форматі 1.7 діапазон можливих значень лежить в межах [0; 2>. Для знакового дробового числа єдиний розряд для цілого значення відведений суто під знак, і тому діапазон можливих значень лежить в межах [-1; 1>.

Таблиця 3.4. Представлення беззнакових дробових чисел 1.7

2-ве	10-ве	1.7	2-ва	10-ве	1.7
00000000	0	0.0	…	…	…
00000001	1	0.0078125	11111101	253	1.9765625
00000010	2	0.015625	11111110	254	1.984375
00000011	3	0.0234375	11111111	255	1.9921875

Таблиця 3.5. Представлення знакових дробових чисел 1.7

2-ве	10-ве		1.7	2-ве	10-ве		1.7
	знак.	беззн.			знак.	беззн.	
00000000	0	0	0.0	10000000	-128	128	-1.0
00000001	1	1	0.0078125	10000001	-127	129	-0.9921875
…	…		…	…	…		…
01111110	126	126	0.984375	11111110	-2	254	-0.015625
01111111	127	127	0.9921875	11111111	-1	255	-0.0078125

Щоб отримати реальне значення числа у форматі 1.7, необхідно його десяткове цілочисельне представлення поділити на число 128.

При множенні чисел у форматі «n.q», наприклад n1.q1× n2.q2, кінцевий результат буде мати представлення (n1+n2).(q1+q2). Однак, для того щоб отриманий результат був у зручному форматі, апаратне множення виконує для результату ще й порозрядний зсув вліво на 1 біт, після чого результат може бути заокруглений до 1.7.

$$1.7\times 1.7 = (2.14) <<1 = 1.15$$

У AVR передбачені три команди апаратного множення 1.7:
- **fmul** → беззнаковий 1.7 × беззнаковий 1.7;
- **fmuls** → знаковий 1.7 × знаковий 1.7;
- **fmulsu** → знаковий 1.7 × беззнаковий 1.7.

Для спрощення формування дробових чисел у форматі 1.7 та 1.15 передбачені спеціальні функції для компілятора AVR (табл. 3.6).

Таблиця 3.6. Додаткові функції компілятора

Q7(*вираз*)	Конвертує число з плаваючою комою до знакового формату 1.7 (для fmul/fmuls/ fmulsu команд)
Q15(*вираз*)	Конвертує число з плаваючою комою до знакового формату 1.15
Int(*вираз*)	Обрізає число з плаваючою комою до цілого числа (відкидає дробову частину)
Frac(*вираз*)	Отримує дробову частину числа з плаваючою комою (відкидає цілу частину)
Abs(*вираз*)	Повертає абсолютне значення з числа

Оскільки функція Q7() формує лише знакове дробове число 1.7, то для отримання беззнакового числа за допомогою цієї функції формуємо дробову частину числа та додаємо 1 у старший розряд.

```
    ldi     r16, Q7(-0.5)           ; r16=192(-64)   → -64/128=-0.5
    ldi     r17, Q7(0.5)            ; r17=64         → 64/128=0.5
    ldi     r18, (1<<7) | Q7(0.5)   ; r18=192        → 192/128=1.5
    ldi     r19, (1<<7) | Q7(0.4)   ; r19=179      → 179/128=1.3984375
    ldi     r20, Q7(-0.9)           ;r20=140(-116) → -116/128=-0.90625

; (беззнаковий 1.7) × (беззнаковий 1.7)
    fmul    r17, r18       ;res=24576      → 24576/32768=0.75
    fmul    r18, r19       ;res=3200       → 3200/32768=0.09765625
                                                       переповнення
; (знаковий 1.7) × (знаковий 1.7)
    fmuls   r16, r17       ;res=57344(-8192) → -8192/32768=-0.25
; (знаковий 1.7) × (беззнаковий 1.7)
    fmulsu  r16, r18       ;res=40960 (-24576) → -24576/32768=-0.75
    fmulsu  r20, r18       ;res=20992          →20992 /32768=0,640625
                                                          переповнення
```

При множенні дробових беззнакових чисел потрібно слідкувати, щоб не було переповнення, тобто результат не виходив за визначені межі (не був більшим за значення 2). Аналогічно, при множенні знакового на беззнаковий також потрібно слідкувати, щоб не було вихід за межі діапазону [-1;1>.

Щоб отримати реальне значення числа у форматі 1.15, необхідно його десяткове цілочисельне представлення поділити на число 2^{15}=32768.

Замість використання функції Q7() ми можемо просто необхідне число з плаваючою комою помножити на 128. Наведений нижче код ідентичний попередньому, з тією лиш різницею, що може бути відмінність в один розряд для чисел, що точно не відображаються на цілочисельний тип та мусять бути заокруглені.

```
ldi     r16, -0.5*128       ; Q7(-0.5)
ldi     r17, 0.5*128        ; Q7(0.5)
ldi     r18, 1.5*128        ; (1<<7) | Q7(0.5)
ldi     r19, 1.4*128        ; (1<<7) | Q7(0.4)
ldi     r20, -0.9*128       ; Q7(-0.9)
```

Якщо ж потрібно множити дробові числа, що виходять за межі, вказані для формату 1.7, наприклад 4.5×6.25, тоді необхідно умовно перейти до іншого формату. Виберемо для цього випадку беззнаковий формат 3.5 з діапазоном значень [0; 8>. У результаті множення беззнакових дробових чисел 3.5 отримаємо результат у форматі 6.10, а якщо врахувати ще зсув при множенні вліво, то кінцевий формат має бути у форматі 5.11.

Для запису у регістр загального призначення нашого числа з плаваючою комою його необхідно помножити на 2^5=32 (у степені кількість розрядів після коми). Для зворотного відображення у число з плаваючою комою необхідно отриманий результат у форматі 5.11 поділити на 2^{11}=2048.

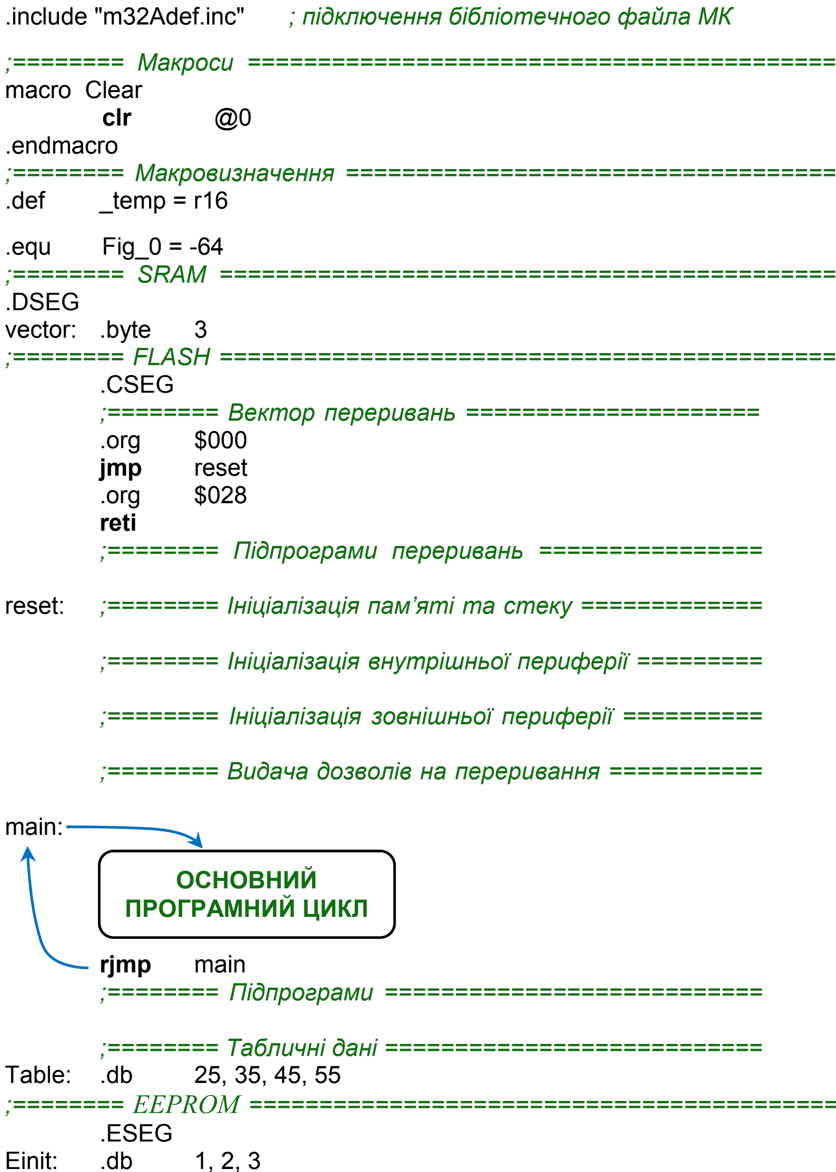
4.5×6.25 = 28.125

```
ldi     r16, 4.5*32         ; r16=144       → 144 / 32 = 4.5
ldi     r17, 6.25*32        ; r17=200       → 200 / 32 = 6.25

; (беззнаковий 3.5) × (беззнаковий 3.5)<<1=(беззнаковий 5.11)
fmul    r16, r17       ;res= 57600     → 57600 / 2048 = 28.125
```

Аналогічні маніпуляції виконуються і при множенні знакових дробових чисел, а також й інших форматів 2.6, 4.4, 5.3, 6.2, 7.1. Формат 8.0 представляє собою цілі числа, і для нього слід використовувати команди апаратного множення mul, muls, mulsu, в яких вже не відбувається зсув вліво, а кінцевий результат представляється у форматі 16.0.

3.4. Структура асемблерної програми.

Для зручного читання коду асемблерної програми, вона повинна мати перш за все чітку структурну організацію, повинна бути швидкою, не здійснювати затримок основного програмного циклу та має легко розширюватися. Ось приблизна структура для такої програми.



```
.include "m32Adef.inc"      ; підключення бібліотечного файла МК

;======== Макроси ==========================================
macro  Clear
          clr         @0
.endmacro
;======== Макровизначення ==================================
.def       _temp = r16

.equ       Fig_0 = -64
;======== SRAM =============================================
.DSEG
vector:   .byte     3
;======== FLASH ============================================
          .CSEG
          ;======== Вектор переривань =====================
          .org      $000
          jmp       reset
          .org      $028
          reti
          ;======== Підпрограми переривань ================

reset:    ;======== Ініціалізація пам’яті та стеку ==========

          ;======== Ініціалізація внутрішньої периферії =====

          ;======== Ініціалізація зовнішньої периферії ======

          ;======== Видача дозволів на переривання ==========

main:
          ОСНОВНИЙ
          ПРОГРАМНИЙ ЦИКЛ
          rjmp      main
          ;======== Підпрограми ===========================

          ;======== Табличні дані =========================
Table:    .db       25, 35, 45, 55
;======== EEPROM ===========================================
          .ESEG
Einit:    .db       1, 2, 3
```

Про певні пункти структури програми ми вже в певній мірі дещо розказували. Тому зупинимося лише на окремих пунктах.

При старті програми МК, а особливо при її рестарті, регістри загального призначення та оперативна пам'ять не завжди є обнуленою. Часто там залишається сміття з попереднього запуску. І тому у програмі необхідно усі комірки пам'яті як оперативної, так і регістри загального призначення або встановлювати в необхідне значення, або обнулювати. Ну хоча б проініціалізувати ті комірки пам'яті, з якими ми будемо працювати.

За замовчуванням при старті МК значення стеку має мати значення кінця SRAM. Але все одно краще проініціалізувати його на початку програми, і бути впевненим, що стек розміщений у визначеному місці.

Ініціалізація внутрішньої периферії передбачає налаштування роботи різних таймерів, інтерфейсів, портів вводу/виводу і т.п.

Ініціалізація зовнішньої периферії передбачає налаштування роботи дисплеїв, зовнішньої пам'яті, ініціалізацію різних пристроїв, давачів, усього, що підключено ззовні до МК.

Коли основна ініціалізація виконана, тоді даються дозволи на переривання від периферії МК та загальний дозвіл (прапорець I регістру стану SREG). Після цього керування передається основному циклу програми.

Структурно підпрограми переривань розміщуються одразу після вектора переривань, а звичайні підпрограми після блоку основного циклу.

В кінці сегменту FLASH розміщуються таблиці з константами.

3.5. Паралельні порти вводу/виводу.

Паралельні порти вводу/виводу є основною складовою будь-якого МК AVR.

Ноги портів, окрім основного призначення – побайтного чи побітного вводу/виводу, можуть також використовуватися для роботи з певною внутрішньою периферією МК (у залежності від того, які альтернативні функції прикріплені до них).

Модель ATmega32A має в себе на борту 4 повних порти вводу/виводу: A, B, C та D. Налаштування та керування портом можемо виконувати як усім одразу, так і кожною його ногою окремо.

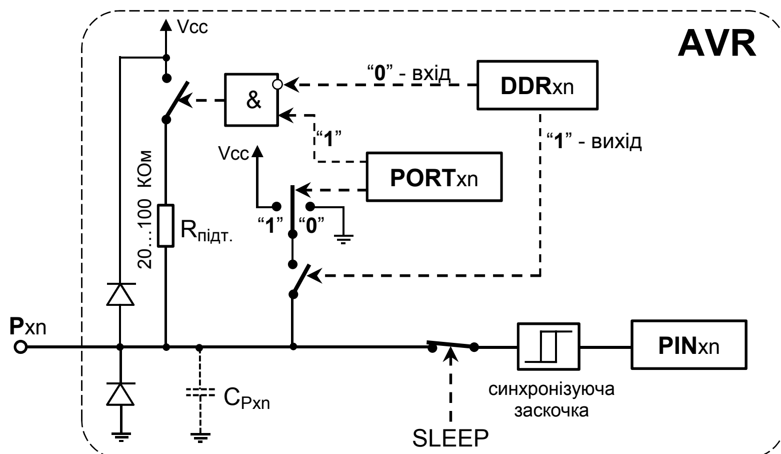


Рис. 3.2. Спрощена функціональна схема ноги порту вводу/виводу без врахування додаткових функцій

Усі ноги портів мають внутрішній діодний захист, який захищає від стрибків підвищеної напруги (верхній діод відкривається при напрузі вищій за напругу живлення МК, і з перепадом напруги вже борються фільтри БЖ) та від'ємної напруги (яка нейтралізується на землю нижнім діодом). Цей діодний захист призначений лише від імпульсних завад, і при перевищеній напрузі нога порту ймовірно вийде з ладу. Варто мати на увазі, що на кожній нозі присутня незначна внутрішня паразитна ємність.

За керування ногами портів МК AVR відповідає набір регістрів DDRx, PORTx та PINx (літера x відповідає назві регістра A, B, C чи D).

DDRx – ці регістри визначають напрямок роботи портів: на вхід чи на вихід. При цьому ми можемо налаштувати індивідуально кожну ногу вибраного порту. Якщо біт у регістрі, що відповідає потрібній нозі порту, дорівнює «0», тоді ця нога працює на вхід, якщо ж «1» – тоді на вихід. Візьмемо, наприклад, порт A:

	7	6	5	4	3	2	1	0
DDRA	1	1	0	0	0	1	1	0
	PA7 (OUT)	PA6 (OUT)	PA5 (IN)	PA4 (IN)	PA3 (IN)	PA2 (OUT)	PA1 (OUT)	PA0 (IN)

PORTx – значення бітів цих регістрів визначають налаштування виводів портів у залежності від значення, встановленого у регістрі DDRx. Якщо вивід працює на вихід, тобто біт у регістрі DDRx встановлений в «1», тоді значення відповідного біта регістра PORTx вказує на рівень напруги на нозі. Значення «1» встановлює на нозі напругу високого рівня (Vcc, напруга живлення МК), значення «0» встановлює низький рівень (цифрова земля).

Якщо вивід працює на вхід, тобто біт у регістрі DDRx встановлений в «0», тоді значення «1» відповідного біта регістра PORTx підключає внутрішній резистор (Rпідт., рис. 3.2), який підтягує цей вивід до напруги живлення МК. Номінал цього підтягуючого резистора є не точним і знаходиться в межах 20-100 КОм. Значення «0» залишає цей вивід у високоімпедансному стані Hi-Z (без підтягуючого резистора), тобто повний опір входу є дуже високим та не впливає на зовнішнє підключення. У стані Hi-Z входи МК використовуються для підключення до сторонніх шин даних, не заважаючи при цьому їхній роботі.

PINx – ці регістри призначені лише для читання та, незалежно від налаштування портів вводу/виводу, завжди відображають поточні стани виводів МК. Для стабільності зчитування станів виводів МК є присутня синхронізуюча ланка, що складається з тригерної заскочки та розряду PINxn (рис. 3.2). Значення сигналу на виводі МК фіксується заскочкою при низькому рівні сигналу та перезаписується потім у розряд PINxn з наростаючим фронтом тактового сигналу. Відповідно, між операціями запису у порт та зчитуванням його стану є присутня затримка. Тому у програмах між командами out та in одного порту необхідно вставляти пустий оператор nop.

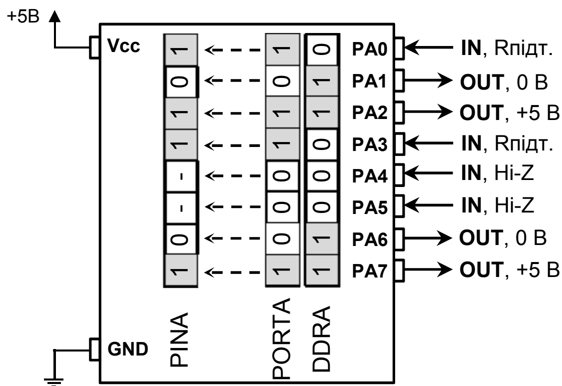


Рис. 3.3. Налаштування виводів порту А та значення регістра PINA

Значення окремих бітів регістрів PINx відповідають реальним рівням сигналів на виводах портів. «1» – при високому рівні, «0» – при низькому рівні. Якщо виводи налаштовані як високоімпедансні входи, та немає підключення зовнішніх сигналів, тоді значення відповідних бітів PINx будуть рівними «0». Однак, від найменших завад, наприклад дотик пальця до корпусу МК, на виході може з'явитися «1». Тому на рис. 3.3 значення бітів для таких виводів мають прочерки «-». Саме по цій причині непідключені входи до шин сигналів необхідно підтягувати через резистор до напруги живлення чи землі.

Поділ за рівнями вхідних зовнішніх сигналів, у залежності від напруги живлення МК, має вигляд як на рис. 3.4.

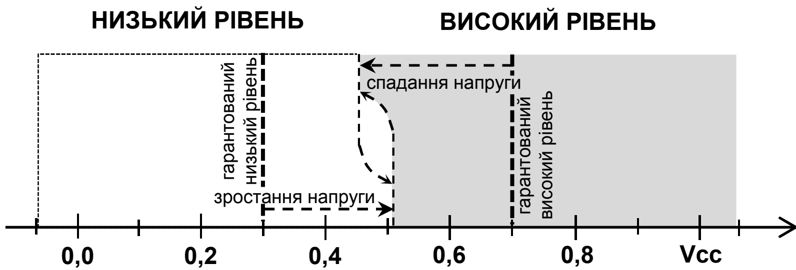


Рис. 3.4. Розподіл рівнів відносно напруги живлення МК з врахуванням гістерезисної петлі тригерної заскочки

Документацією визначені гарантовані рівні сигналів:

- від -0,5В до 0,3·Vcc – низький рівень;
- від 0,7·Vcc до Vcc+0.5В – високий рівень.

Нижчі чи вищі за вказані рівні напруг можуть вивести з ладу вивід МК. На границі рівнів (рис. 3.4) присутня так звана гістерезисна петля. Перехід від низького рівня до високого відбувається при одній напрузі, а зворотній перехід, від високого до низького рівня, при дещо нижчій напрузі. Ця гістерезисна петля є своєрідним механізмом захисту від переключення рівнів під дією завад. У документації розмах величини гістерезисної петлі визначений 0,05·Vcc. Середня ділянка рівнів дещо відрізняється для різних моделей та партії випуску.

Експериментальні зняття показів з декількох ATmega32A при напрузі живлення +5 В встановили такий усереднений діапазон розмаху гістерезисної петлі: 2,36-2,52 В. Тобто, перехід від низького до високого рівня встановлюється при напрузі 2,52 В, а від високого до низького при 2,36 В.

При підключенні різноманітних кнопок та клавіатур до МК для уникнення дії завад необхідно задіювати підтягуючі резистори до напруги живлення чи землі. Найпростіше налаштувати вхід на використання внутрішнього підтягуючого резистора до напруги живлення.

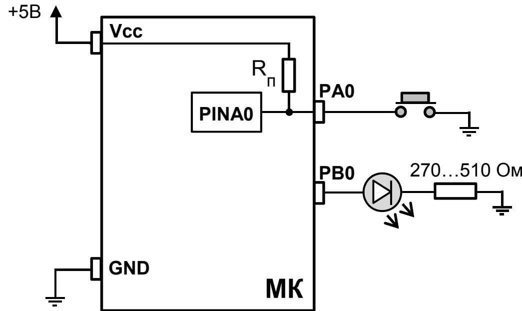


Рис. 3.5. Підключення кнопки та світлодіода до МК

Покажемо на прикладі (рис. 3.5) програмну реалізацію підключення кнопки та світлодіода до МК.

```
.include "m32Adef.inc"

        .CSEG
        ldi     r16, 0x00
        ldi     r17, 0xFF
        ;Порт A на вхід з підтягуючим резистором
        out     DDRA, r16
        out     PORTA, r17
        ;Порт B на вихід з низьким початковим рівнем
        out     DDRB, r17
        out     PORTB, r16

main:
        in      r16, PINA
        out     PORTB, r16
        rjmp    main
```

На початку програми ми виконуємо налаштування портів, до яких підключені зовнішні пристрої. Кнопка під'єднана до нульового виводу порту A, світлодіод до нульового виводу порту B. Увесь порт A ми налаштували на вхід та включили внутрішні підтягуючі резистори. Порт B налаштували на вихід. В основному програмному циклі ми зчитуємо стан входів усього порту A та заносимо цей байт стану у

регістр загального призначення r16. На наступному такті програми значення r16 виводимо на виходи порту B. Якщо кнопка не натиснута, тобто на вхід не під'єднана зовнішня земля, то стан виводу має значення «1», оскільки внутрішні підтягуючі резистори подають високий рівень. Натисканням кнопки ми шунтуємо підтягуючий резистор, подаючи безпосередньо землю на вхід, і отримуємо значення «0».

Електричні характеристики системи вводу-виводу. Максимальне абсолютне значення допустимого струму для однієї ноги системи вводу/виводу складає 40 мА. Але рекомендованим є значення, що не перевищує 20 мА при Vcc = 5 В та 10 мА при Vcc = 3 В. Однак загальний струм, що протікає через усі виводи МК не повинен перевищувати:

- 200 мА для МК AVR у DIP корпусі;
- 400 мА для МК AVR у TQFP та QFN корпусах.

Також обмеження накладаються на окремі порти. Для моделі ATmega32A ситуація є такою:

- DIP корпус (200 мА):
  - 100 мА для порту A;
  - 100 мА для решти B, C, D портів.
- TQFP чи QFN корпус (400 мА):
  - 100 мА для порту A;
  - 100 мА для виводів B0-B4
  - 100 мА для виводів B3-B7 та D0-D2;
  - 100 мА для виводів D3-D7;
  - 100 мА для порту C.

Отже при підключенні пристроїв до портів вводу/виводу необхідно чітко прораховувати кількість струму, що буде протікати через ноги МК та через які саме.

Оскільки виводи МК AVR забезпечують високу навантажувальну здатність при любому рівні сигналу, то до них можна безпосередньо підключати світлодіодну індикацію.

Вольт-амперна характеристика світлодіода така ж як у звичайних діодів (рис. 3.6). Для уникнення виходу з ладу ноги МК та світлодіода необхідно виконувати підключення світлодіодів через обмежувальні резистори. Визначення опору цього резистора здійснюють, виходячи зі струму, який має споживати світлодіод. Наприклад, для нашого FYL-3014LURC1A виберемо робочий струм 10 мА. На ВАХ цьому струму відповідає напруга 2,1 В.

$$R_{ОБМЕЖ} = (V_{CC} - V_{LED}) / I_{LED} = (5 - 2{,}1) / 0{,}01 = 290 \text{ Ом}$$

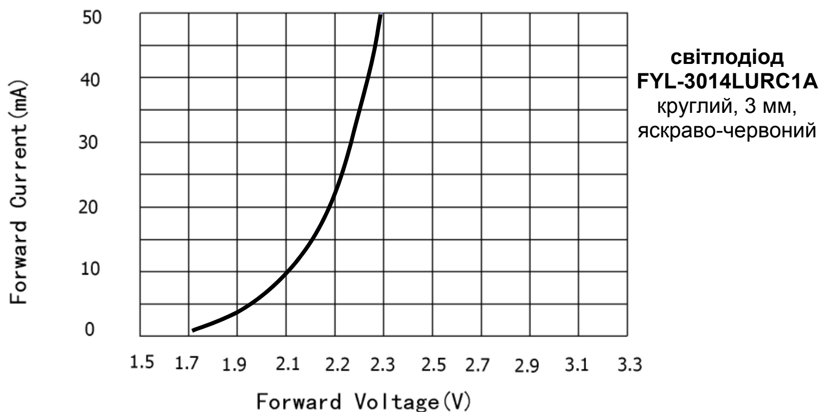


Рис. 3.6. Вольт-амперна характеристика світлодіода

3.6. Варіанти підключення кнопок та світлодіодів до МК.

При підключенні світлодіодів необхідно чітко дотримуватися полярності: високий потенціал (+) до анода діода, низький потенціал (–) до катода. Відповідно до цього, ми можемо по різному підключати світлодіоди: або до зовнішньої додатної напруги та засвічувати світлодіод низьким рівнем («0») з виходу МК, або до зовнішньої землі та засвічувати високим рівнем з виходу МК (рис. 3.7 а). Вибір одного з цих двох способів підключення найчастіше обумовлюється зручністю підведення землі чи додатної напруги до світлодіода.

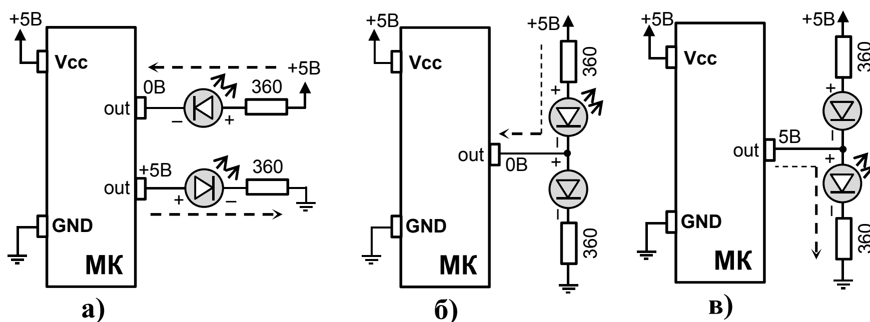


Рис. 3.7. Підключення світлодіодів

Для деяких задач необхідно почергово засвічувати два світлодіоди, наприклад, червоний світлодіод – канал передає дані, зелений –

канал відключений. Тоді їх можемо підключити разом до одного виводу МК, як на рис. 3.7. Подавши «0» на вихід – засвітимо верхній світлодіод (б), подавши «1» – засвітимо нижній (в). Якщо цей вивід МК налаштуємо як високоімпедансний вхід, тоді струм потече від додатної напруги до землі через обидва світлодіоди, і вони будуть обидва світитися, але дещо тьмяніше.

Для забезпечення завадостійкості при підключенні кнопок до виводів МК необхідно використовувати зовнішній підтягуючий резистор, оскільки у деяких випадках внутрішній резистор може не справлятися із наявними завадами. При цьому вивід можемо підтягувати як до напруги живлення, так і до землі. Відповідно, кнопки мають бути підключеними до землі та напруги живлення (рис. 3.8 а). У другому випадку (з підтягуючим резистором до землі) логіка входу є прямою: при відпущеній кнопці – «0», при натисненій – «1».

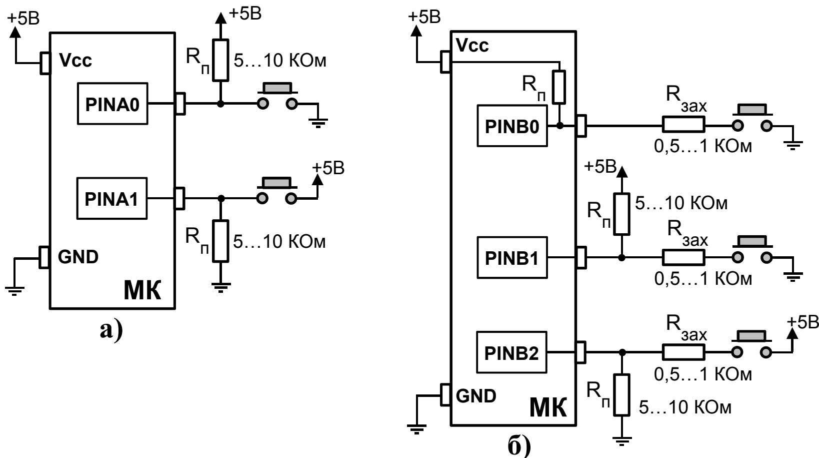


Рис. 3.8. Підключення кнопок

Для запобігання аварій через невірне налаштування виводів портів (на вихід, замість того, щоб на вхід), до яких підключені кнопки, необхідно підключати послідовно з кнопками захисні резистори (рис. 3.8 б).

Існує чимало різноманітних варіацій підключення кнопок та світлодіодів до виводів МК: 3 кнопки до 2 ніг, 6 чи 7 кнопок до 3 ніг, кнопка та світлодіод на 1 нозі, багато кнопок на 1 вхід АЦП і т.п.

3.7. Умовні та безумовні переходи та регістр стану SREG.

МК AVR мають три **команди безумовного переходу**:

jmp – абсолютний (прямий) перехід (direct jump);
rjmp – відносний перехід (relative jump);
ijmp – непрямий перехід (indirect jump to (Z)).

Покажемо на прикладі виконання команд безумовного переходу.

```
.include "m32Adef.inc"
        .CSEG
        .org    $085            ;встановлення програмної адреси
main:   nop                     ;пустий оператор
        ldi     r16, 0x25       ;завантаження в r16 значення
        jmp     main            ;абсолютний перехід

        nop                     ;пустий оператор
        rjmp    main            ;відносний перехід

        ldi     ZL, Low(main)   ;завантаження в регістрову пару Z
        ldi     ZH, High(main)  ;адреси переходу
        ijmp                    ;непрямий перехід
```

Згенерований дисасемблером код програми

адреса у пам'яті програм	код команди	назва команди	аргументи	коментар
+00000085:	0000	NOP		No operation
+00000086:	E205	LDI	R16,0x25	Load immediate
+00000087:	**940C0085**	**JMP**	**0x00000085**	**Jump**
+00000089:	0000	NOP		No operation
+0000008A:	**CFFA**	**RJMP**	**PC-0x0005**	**Relative jump**
+0000008B:	E8E5	LDI	R30,0x85	Load immediate
+0000008C:	E0F0	LDI	R31,0x00	Load immediate
+0000008D:	**9409**	**IJMP**		**Indirect jump to (Z)**

Для команди абсолютного переходу **jmp** ми задаємо в її аргументі числове значення адреси у FLASH, туди, куди хочемо виконати перехід. Найпростіше у потрібному місці коду програми поставити мітку і вказати її в аргументі команди. Тоді компілятор собі сам визначить значення адреси, згідно вказаної мітки. У наведеному

прикладі ми переходимо командою jmp на адресу мітки main і, якщо зазирнути у згенерований дисасемблером код, ми бачимо, що там вже підставлене значення 0x00000085, яке відповідає адресі стрічки, на яку посилається мітка. Код команди jmp займає у програмі 4 байти (32 біти): 10 біт під саму команду переходу та 22 біти під адресу, куди має виконатися перехід. Команда jmp може адресувати у будь-яку точку програми, обсягом до 4M слів (макс. адреса 0x3FFFFF). Для усіх моделей AVR розмір FLASH наразі не має таких великих об'ємів. 2 байтів достатньо для адресації 128 Кбайтів FLASH. Варто зазначити, що старші біти адреси є частково перемішані з бітами коду команди, але 2 молодші байти є розташовані разом. Тому в дисасемблері ми бачимо загальний код для команди переходу з вказаною адресою як 940C0085, де 0085 і є адресою переходу. Команда jmp виконується за 3 такти МК.

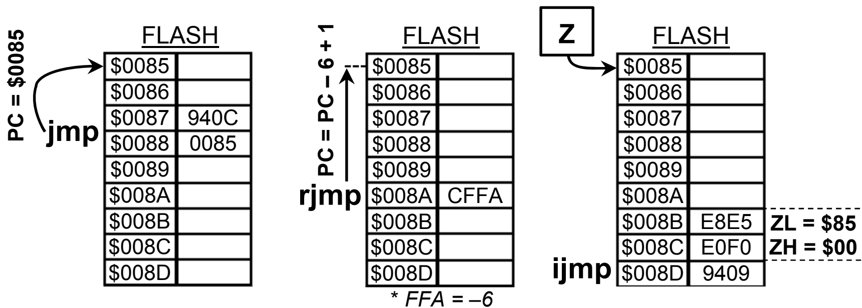


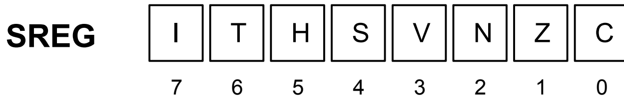
Рис. 3.9. Ілюстрація роботи команд для безумовного переходу

Команда відносного переходу **rjmp** виконує перехід, здійснюючи зміщення вперед чи назад у програмній пам'яті на вказане число відносно значення поточної адреси (програмного лічильника). Код команди зі значенням зміщення займає 2 байти (16 біт), 12 бітів з яких відведені під значення зміщення. Тому значення зміщення адреси програмного лічильника може приймати значення -2047…+2048 у пам'яті програм. У програмному коді як аргументи вказують мітки переходу. Компілятор сам обрахує необхідне значення величини зміщення та підставить у слово команди. Тому великої різниці між командами jmp та rjmp немає, лише в обмеженому діапазоні адресації переходу. Зате команда rjmp виконується за 2 такти.

Команда непрямого переходу **ijmp** виконує перехід за адресою, що розміщена в індексному регістрі Z. Код команди ijmp займає у програмі 2 байти, а команда виконується за 2 такти МК. Оскільки

індексний регістр Z двобайтний, то об'єм FLASH для цієї команди обмежується 128 кБайтами. Завдяки команді ijmp ми можемо програмно змінювати нашу точку переходу, що робить програму гнучкішою.

Перед розглядом команд умовного переходу розберемо призначення окремих прапорців **регістру стану SREG**. Ці прапорці встановлюються у залежності від результату виконання арифметичних, логічних та порозрядних операцій над регістрами загального призначення в ALU.



C – прапорець переносу вказує на перенос чи позику при виконанні арифметичної чи логічної операції, тобто тоді, коли відбувся вихід за межі байта.

Z – прапорець нуля вказує на нульовий результат при виконані арифметичної чи логічної операції.

N – прапорець від'ємного результату вказує на від'ємний результат при виконані арифметичної чи логічної операції. Тобто в старшому розряді присутня «1». Цей прапорець має на увазі, що ми працюємо зі знаковими числами в діапазоні від -128 до 127. Ми вже раніше звертали увагу на те, що числа -25 та 231 для 8-ми розрядного асемблера є ідентичними, оскільки у двійковому представленні вони виглядають однаково: 0b11100111.

V – прапорець переповнення у доповняльному коді, тобто коли відбувається вихід за межі [-128; 127].

S – прапорець знаку є результатом суми за модулем 2 (виключне або) між прапорцями N та V ($S=N \oplus V$). Він з'являється, якщо при обчисленні чисел зі знаком результатом є від'ємне число, і при цьому не було переповнення у доповняльному коді, або отримуємо додатне число у результаті переповнення у доповняльному коді. В основному він використовується при порівнянні знакових чисел.

H – прапорець часткового переносу вказує на перенос чи позику між старшою половиною байта та молодшою при виконанні деяких арифметичних операцій. Частковий перенос є корисний у двійково-десятковій арифметиці.

```
ldi     r16, 120          ;r16=120
ldi     r17, 150          ;r17=150=-106
add     r16, r17          ;r16=120+150=120-106=14        |  |  |H|S|V|N|Z|C|
subi    r16, 20           ;r16=14-20=-6=250              |  |  |H|S|V|N|Z|C|

ldi     r16, 0b0000_0111          ;r16=7
ldi     r17, 0b0000_1100          ;r17=12
sub     r16, r17          ;r16=7-12=-5=251=0b1111_1011   |  |  |H|S|V|N|Z|C|
```

T – користувацький біт для роботи з бітами регістрів загального призначення у командах bld (bit load) та bst (bit store). Команда bst копіює значення вказаного біта із зазначеного регістра загального призначення у біт T регістра стану SREG. За допомогою команди bld можна навпаки, з біта T встановити значення у вказаному біті зазначеного регістра загального призначення. Це є стандартний підхід для встановлення/скинення необхідного біта у регістрі загального призначення. Попереднього біт T встановлюється/скидається за допомогою призначених для цього команд set/clt.

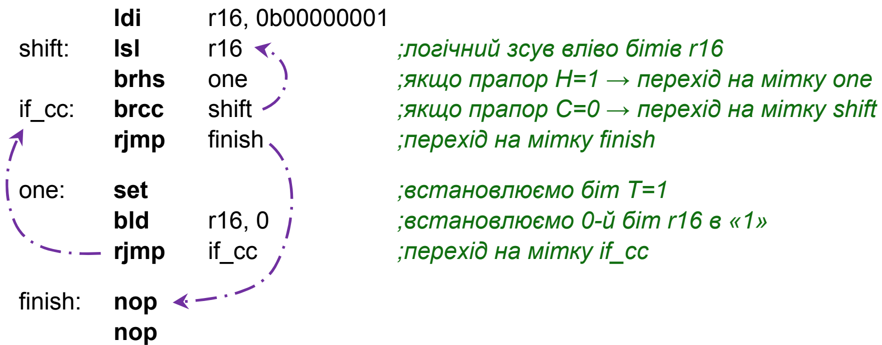
```
set               ;встановлюємо біт T                    |  |T|  |  |  |  |  |  |
bld     r20, 5    ;встановлюємо 5-й біт у r20

clt               ;очищуємо біт T                        |  |T|  |  |  |  |  |  |
bld     r20, 5    ;очищуємо 5-й біт у r20
```

I – прапорець дозволу глобальних переривань. Коли він встановлений в «0» – будь-які переривання призупинені, коли в «1» – дозволені переривання знову активні. Команда sei – надає загальний дозвіл на переривання, cli – забороняє будь-які переривання.

Кожен з цих прапорців регістру стану SREG ми також можемо встановити/скинути за допомогою призначених для цього команд. Для прапорця C – sec/clc; для Z – sez/cln; для N – sen/cln і т.д.

Команди умовного переходу виконують перехід, опираючись на значення одного з бітів регістра стану SREG. Тобто, для кожного прапорця стану є визначені свої команди переходу, у залежності від його значення. Якщо визначений прапорець відповідає зазначеній умові (встановлений чи обнулений), відбувається перехід у визначене місце у програмі, якщо ні – виконується наступна команда. Ці всі команди утворюють групу Branch, і мають однаковий принцип дії.

Назва команди розпочинається 2-ма літерами br$_{xx}$, а дві останні літери команди визначають логіку переходу.

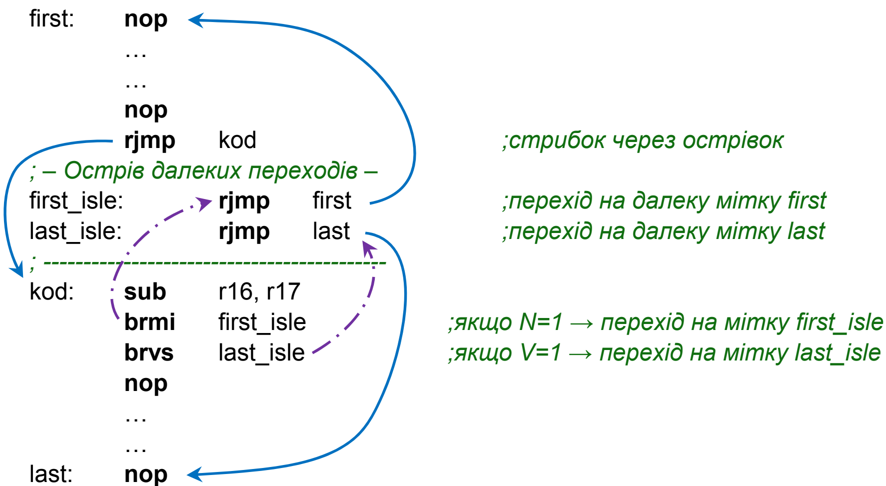


```
          ldi     r16, 0b00000001
shift:    lsl     r16             ;логічний зсув вліво бітів r16
          brhs    one             ;якщо прапор H=1 → перехід на мітку one
if_cc:    brcc    shift           ;якщо прапор C=0 → перехід на мітку shift
          rjmp    finish          ;перехід на мітку finish

one:      set                     ;встановлюємо біт T=1
          bld     r16, 0          ;встановлюємо 0-й біт r16 в «1»
          rjmp    if_cc           ;перехід на мітку if_cc

finish:   nop
          nop
```

команди	*r16*								*SREG*						
ldi r16,1	0	0	0	0	0	0	0	1		H	S	V	N	Z	C
lsl r16	0	0	0	0	0	0	1	0		H	S	V	N	Z	C
lsl r16	0	0	0	0	0	1	0	0		H	S	V	N	Z	C
lsl r16	0	0	0	0	1	0	0	0		H	S	V	N	Z	C
lsl r16	0	0	0	1	0	0	0	0		H	S	V	N	Z	C
set									T						
bld r16,0	0	0	0	1	0	0	0	1	T	H	S	V	N	Z	C
lsl r16	0	0	1	0	0	0	1	0		H	S	V	N	Z	C
lsl r16	0	1	0	0	0	1	0	0		H	S	V	N	Z	C
lsl r16	1	0	0	0	1	0	0	0		H	S	V	N	Z	C
lsl r16	0	0	0	1	0	0	0	0		H	S	V	N	Z	C
set									T						
bld r16,0	0	0	0	1	0	0	0	1	T	H	S	V	N	Z	C
finish															

Рис. 3.10. Ілюстрація роботи умовних переходів

Команди переходу групи Branch подібні до команди rjmp тим, що виконують перехід, здійснюючи зміщення вперед чи назад у програмній пам'яті на вказане число відносно значення поточної адреси (програмного лічильника). Коди команд групи Branch займають 2 байти (16 бітів), 7 бітів з яких відведені під значення зміщення, яке, відповідно, може приймати значення -64…63. Команди групи Branch виконують переходи на близькі віддалі, і тому для збільшення їхньої дистанції розміщують неподалік посеред програмного коду проміжні острівки з командами jmp чи rjmp для переходу на віддалені адреси у пам'яті програм. Команди групи Branch виконуються за 1 такт, якщо умова не справджується, і продовжує виконання наступна в програмі команда, за 2 такти, якщо виконується перехід.



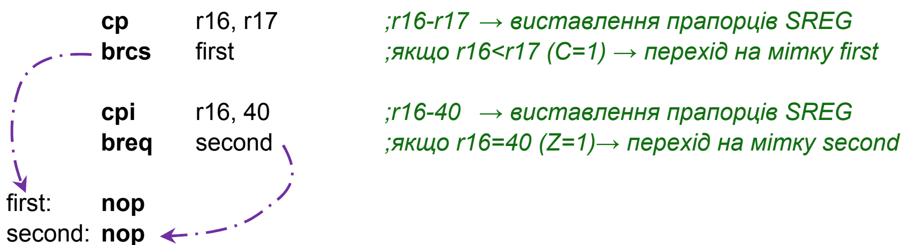
Для організації умов для команд умовного переходу, наприклад чи рівні між собою 2 регістри загального призначення, використовуються **команди групи Compare**. Їхня дія подібна команді віднімання: вони віднімають одне значення від іншого, виставляють прапорці регістра стану SREG, але отриманий числовий результат нікуди не заносять. Таким чином, ми можемо виконати перехід Branch на основі порівняння двох значень: вони рівні, одне більше за інше і т.п. Асемблер містить 4 команди групи Compare:

cp – порівняння (compare) → виставляє прапорці на основі результату Rd–Rr;

cpc – порівняння з врахуванням переносу (compare with carry) → виставляє прапорці на основі результату Rd–Rr–C;

cpi – порівняння регістра з константою (compare register with Immediate) → виставляє прапорці на основі результату Rd–K;

cpse – порівнює Rd=Rr, якщо вони рівні, тоді пропускає наступну команду (compare, skip if equal);



Перші три команди групи Compare виконуються за 1 такт, остання команда cpse – за 1 такт, якщо умова не виконується, а якщо виконується, тоді: 2 такти, при розмірі команди, що пропускається, 1 слово, та 3 такти, при розмірі команди, що пропускається, 2 слова.

На основі поєднання команд Compare та Branch можна утворювати **умовні конструкції** типу if(умова) … else, як у мові Сі.

Таблиця 3.7. Умовні конструкції на основі команди cp

Умова	**Знакові числа** [-128; 127]		**Беззнакові числа** [0; 255]	
	SREG	**Команди**	**SREG**	**Команди**
Rd = Rr	Z = 1	**cp** Rd, Rr **breq** <мітка>	Z = 1	**cp** Rd, Rr **breq** <мітка>
Rd ≠ Rr	Z = 0	**cp** Rd, Rr **brne** <мітка>	Z = 0	**cp** Rd, Rr **brne** <мітка>
Rd < Rr	S = 1	**cp** Rd, Rr **brlt** <мітка>	C = 1	**cp** Rd, Rr **brcs** <мітка>
Rd >= Rr	S = 0	**cp** Rd, Rr **brge** <мітка>	C = 0	**cp** Rd, Rr **brcc** <мітка>
Rd > Rr	Z = 0 та S = 0	**cp** Rd, Rr **breq** next **brge** <мітка> next:	Z = 0 та C = 0	**cp** Rd, Rr **breq** next **brcc** <мітка> next:
*альтерн.ум. (Rr < Rd)***	*(S = 1)***	***cp*** *Rr, Rd* ***brlt*** *<мітка>*	*(C = 1)***	***cp*** *Rr, Rd* ***brcs*** *<мітка>*
Rd <= Rr	Z = 1 або S = 1	**cp** Rd, Rr **breq** <мітка> **brlt** <мітка>	Z = 1 або C = 1	**cp** Rd, Rr **breq** <мітка> **brcs** <мітка>
*альтерн.ум. (Rr >= Rd)***	*(S = 0)***	***cp*** *Rr, Rd* ***brge*** *<мітка>*	*(C = 0)***	***cp*** *Rr, Rd* ***brcc*** *<мітка>*

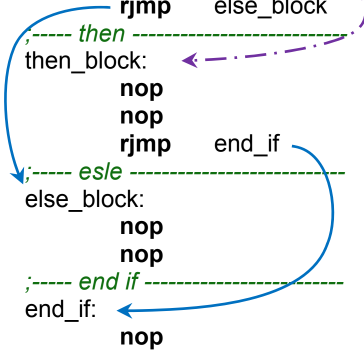
* $S = N \oplus V$

** *Для переходу за цією умовою операнди команди порівняння повинні бути записані у зворотному порядку, тобто замість cp Rd, Rr → cp Rr, Rd*

Варто мати на увазі, що умови порівняння для знакових та беззнакових типів використовують різні прапорці регістру стану SREG. Наприклад, порівнюючи числа -46(210) та 36(36), у нас перше число як знакове є меншим, бо -46 < 36, але як беззнакове воно більше, бо 210 > 36.

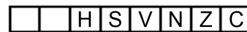
Покажемо на прикладі асемблерну реалізацію умовної конструкції з умовою (r16<r17), де вхідні числа є знаковими. Для цього виберемо з таблиці 3.7 необхідний нам варіант.

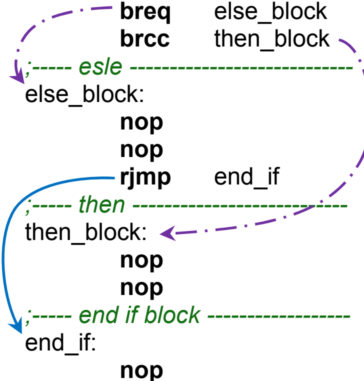
Rd < Rr	S = 1	**cp** Rd, Rr **brlt** <мітка>



```
        ldi     r16, -46
        ldi     r17, 36
;----- if(r16<r17) ---знакові-------
        cp      r16, r17
        brlt    then_block    ;якщо r16<r17 → перехід на мітку then_block
        rjmp    else_block
;----- then ---------------------------
then_block:
        nop
        nop
        rjmp    end_if
;----- esle ---------------------------
else_block:
        nop
        nop
;----- end if -------------------------
end_if:
        nop
```

У табл. 3.7 дві останні умови мають альтернативні варіанти рішення за рахунок перестановки операндів. Однак для команди cpi (порівняння регістра з константою) таку перестановку зробити не можна. У такому випадку слід реалізовувати основний варіант.





```
        ldi     r16, 55
;----- if(r16>const) --беззнакові-
        cpi     r16, 50
        breq    else_block    ;якщо r16=K → перехід на мітку else_block
        brcc    then_block    ;якщо r16>=K → перехід на мітку then_block
;----- esle ---------------------------
else_block:
        nop
        nop
        rjmp    end_if
;----- then ---------------------------
then_block:
        nop
        nop
;----- end if block -----------------
end_if:
        nop
```

Ще одним різновидом умовних команд є **команди групи Skip**. Вони призначені для перевірки окремих бітів регістрів загального призначення R0-R31 та молодших 32 регістрів вводу виводу з адресами у межах $0-$1F (Це регістри портів A, B, C та D, SPI, TWI, EEPROM; деякі з регістрів USART, АЦП, аналогового компаратора).

Принцип роботи команд групи Skip полягає у перевірці вказаних бітів у зазначених регістрах, і якщо відповідний біт є встановлений чи очищений, у залежності від умови команди skip, тоді наступна команда пропускається. Є 4 таких команди:

sbrc – пропустити наступну команду, якщо вказаний біт в регістрі заг. призначення дор. «0» (skip if bit in register cleared);

sbrs – пропустити наступну команду, якщо вказаний біт в регістрі заг. призначення дор. «1» (skip if bit in register is set);

sbic – пропустити наступну команду, якщо вказаний біт в регістрі вводу/виводу дор. «0» (skip if bit in I/O register cleared);

sbis – пропустити наступну команду, якщо вказаний біт в регістрі вводу/виводу дор. «1» (skip if bit in I/O register is set).

Перед тим ми ще розглядали команду cpse з групи Compare, яка за своєю дією подібна до них.

Якщо команди Compare та Branch використовувалися для порівняння числових значень, то команди Skip використовуються для організації логіки на основі значень окремих бітів. Наприклад, ми можемо на основі одного з регістрів загального призначення організувати собі регістр програмних прапорців.

```
.def    _flags = r10        ; _flags  | f7 | f6 | f5 | f4 | f3 | f2 | f1 | f0 |
        .CSEG                            7    6    5    4    3    2    1    0
        clt
        bld     _flags, 0
        set
        bld     _flags, 1   ; _flags  | f7 | f6 | f5 | f4 | f3 | f2 | f1 | f0 |
        …                                7    6    5    4    3    2    1    0
;---------- if (f0==1) ----------------
        sbrs    _flags, 0
        rjmp    f0_end
;---------- then ----------------------
        nop
f0_end:
;---------- if (f1==1) ----------------
        sbrs    _flags, 1
        rjmp    f1_end
;---------- then ----------------------
        nop
f1_end:
```

Приклад1. Розрахунок програмної затримки. Для реалізації найпростішої затримки завантажуємо процесор МК циклічною роботою з умовою та рахуємо такти. Найлегше виконувати віднімання 1 від якогось числа та перевіряти на нульовий результат. Якщо необхідно більшу затримку, тоді задіюють декілька регістрів та використовують операції віднімання з врахуванням переносу.

```
        ldi     r16, 2          ; 1 такт
        ldi     r17, 2          ; 1 такт

delay:  subi    r16, 1          ; 1 такт
        sbci    r17, 0          ; 1 такт
        brne    delay           ; 2 такт (1 – коли нема переходу)
        nop
```

Таблиця 3.8. Покрокова трансляція програми затримки

	команди	R16	R17	прапорці		такти	
	ldi r16,2	2		Z	C	1	1
	ldi r17,2	2	2	Z	C	1	2
delay:	**subi** r16,0	1	2	Z	C	1	3
	sbci r17,0	1	2	Z	C	1	4
	brne delay			Z	C	2	6
delay:	**subi** r16,0	0	2	Z	C	1	7
	sbci r17,0	0	2	Z	C	1	8
	brne delay			Z	C	2	10
delay:	**subi** r16,0	255	2	Z	C	1	11
	sbci r17,0	255	1	Z	C	1	12
	brne delay			Z	C	2	14
	…	254…2					…
delay:	**subi** r16,0	1	1	Z	C	1	1027
	sbci r17,0	1	1	Z	C	1	1028
	brne delay			Z	C	2	1030
delay:	**subi** r16,0	0	1	Z	C	1	1031
	sbci r17,0	0	1	Z	C	1	1032
	brne delay			Z	C	2	1034
delay:	**subi** r16,0	255	1	Z	C	1	1035
	sbci r17,0	255	0	Z	C	1	1036
	brne delay			Z	C	2	1038
	…	254…2					…
delay:	**subi** r16,0	1	0	Z	C	1	2051
	sbci r17,0	1	0	Z	C	1	2052
	brne delay			Z	C	2	2054
delay:	**subi** r16,0	0	0	Z	C	1	2055
	sbci r17,0	0	0	Z	C	1	2056
	brne delay			Z	C	1	2057
	nop						

У регістрах R16 та R17 ми записали число 0x0202 (514). Один цикл віднімання займає у нас 4 такти. Отже розрахунок отримується так:

$$SumCycle = Ncycle \times Value - 1 + Nldi$$

де Cycle – кількість тактів у циклі; Value – число, записане у регістрах; Nldi – кількість команд ldi для запису значень у регістри; SumCycle – сумарна кількість тактів затримки.

$$SumCycle = 4 \times 514 - 1 + 2 = 2057$$

Зворотна задача, визначення необхідного числа, виглядає так:

$$Value = (SumCycle - Nldi + 1) / Ncycle$$

Обчислимо необхідну кількість тактів для 1 мсек:

$$SumCycle = час * частоту\ тактування = 0{,}001 * 8*10^6 = 8000$$

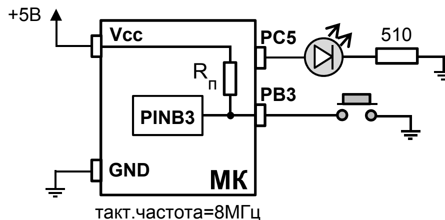
$$Value = (8000 - 2 + 1) / 4 = 1999{,}75 = \sim 2000 = 0x07D0$$

Оскільки ми виконали заокруглення, то реальна кількість тактів становить 8001, а тривалість затримки 0,001000125 сек., що є цілком прийнятно.

Таблиця 3.9. Вибір кількості байтів для розрахунку тривалості

К-сть байтів	Макс. к-сть тактів	Макс. тривалість
1	765	95,625 мкСек
2	262 141	~32,76 мСек
3	83 886 077	~10,48 Сек
4	25 769 803 773	~3 221 Сек

Приклад2. Блимаючий світлодіод. У пасивному режимі світлодіод блимає з інтервалом приблизно 1 сек., тобто 1 сек. світиться, а на 1 сек. гасне. При натисненні кнопки, світлодіод починає блимати у 2 рази частіше.



Розрахуємо значення числа для затримки в 1 сек. з використанням 3 регістрів загального призначення.

$$Value_{1сек} = (8\cdot10^6 - 3 + 1)/5 = \sim1{,}6\cdot10^6 = 0x186A00$$

Значення числа для затримки в 0,5 сек. буде у 2 рази меншим

$$Value_{0{,}5сек} = 0{,}8\cdot10^6 = 0x0C3500.$$

```
.include "m32Adef.inc"
.def        _temp1 = r16
.def        _temp2 = r17
.def        _temp3 = r18

            .CSEG
            ldi         _temp1, 0x00
            ldi         _temp2, 0xFF
            ;Порт B на вхід з підтягуючим резистором
            out         DDRB, _temp1
            out         PORTB, _temp2
            ;Порт C на вихід
            out         DDRC, _temp2
            out         PORTC, _temp1
main:
            ;Інверсія виходу PC5 світлодіода
            sbic        PORTC, 5
            rjmp        elsePC5
            sbi         PORTC, 5
            rjmp        skipPC5end
elsePC5:
            cbi         PORTC, 5
skipPC5end:
            ;Вибір значення затримки в залежності чи натиснута кнопка PB3
            sbis        PINB, 3
            rjmp        elsePB3
            ldi         _temp1, 0x00
            ldi         _temp2, 0x6A
            ldi         _temp3, 0x18
            rjmp        delay
elsePB3:
            ldi         _temp1, 0x00
            ldi         _temp2, 0x35
            ldi         _temp3, 0x0C

delay:      subi        _temp1, 1
            sbci        _temp2, 0
            sbci        _temp3, 0
            brne        delay

            rjmp        main
```

3.8. Використання стеку.

Стек розміщується в SRAM. Він представляє собою LIFO-буфер (last input first output, останнім увійшов – першим вийшов). Це як колода карт, зверху кладете і зверху ж берете. Початок стека розміщується унизу SRAM і по мірі його заповнення він росте в напрямку початку SRAM. Керування стеком виконується за допомогою спеціального 2-байтного регістра SP, що розміщується у 2-х регістрах SPH (старший байт) та SPL (молодший байт). Деякі молодші моделі мають в наявності лише SPL. Значення стеку за замовчуванням рівне 0x0000, тому стек необхідно проініціалізувати вручну. Звісно вибір початку розміщення вершини стеку може бути довільним, однак, якщо його розмістити доволі близько до оперативних даних, тобто ближче до початку SRAM, тоді, по мірі збільшення стеку, може бути його «наїзд» на дані у SRAM, що призведе до їх спотворення та до не правильної роботи програми. Тому стек розміщують у кінці SRAM.

```
ldi     r16, Low(RAMEND)
out     SPL, r16
ldi     r16, High(RAMEND)
out     SPH, r16
```

Значення константи RAMEND залежить від моделі МК та зберігається у конфігураційному файлі цієї моделі.

Стек неявно використовується підпрограмами через команди call, rcall, icall, ret, reti та перериваннями для збереження зворотної адреси точки виклику в програмі. Також стек можна використовувати для швидкого зберігання байтів інформації. Наприклад, нам необхідно регістр для виконання певних операцій, а вільного немає. Тоді вибираємо один з регістрів, що на даний момент не використовується, зберігаємо його значення у стек, працюємо з ним, а по закінченню маніпуляцій з ним відновлюємо зі стеку його попереднє значення.

Для роботи зі стеком призначені дві команди:

push – заносить значення регістра загального призначення у стек (push register on stack), 2 такти;

pop – витягує значення зі стеку у регістр загального призначення (pop register from stack), 2 такти;

Ці команди працюють лише через регістри загального призначення, і якщо необхідно зберегти у стеку значення регістра вводу/виводу, то його перед тим необхідно скопіювати у регістр загального призначення.

При використанні стеку необхідно чітко контролювати логіку роботи з ним, та особливо, щоб не трапився неконтрольований збій у його роботі. Найпростіший випадок, це якщо ми запхали дані в одному порядку, а витягнули не в тому, якому б хотіли. Неконтрольований збій – це якщо ми запхали дані у стек і забули витягнути, або навпаки, витягнули більше, аніж запхали. Тоді відбувається зміщення лічильника стеку, і ми можемо, наприклад, своїми діями переписати збережені адреси повернення підпрограми, чи, якщо ця помилка трапляється циклічно, запороти усі дані в SRAM.

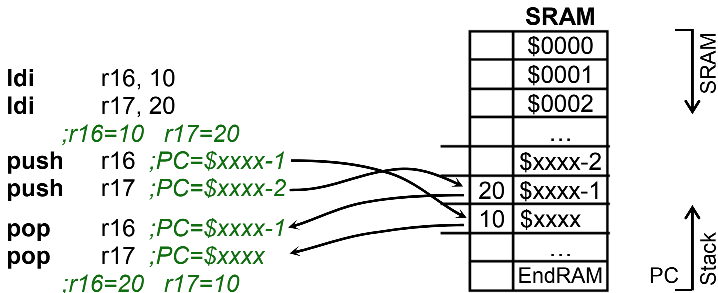


Рис. 3.11. Обмін двох регістрів значеннями через стек

На рис. 3.11 наведено приклад явного використання стеку для обміну регістрами між собою значеннями без використання третього проміжного регістра.

3.9. Підпрограми.

Підпрограми це окремо виділені куски програмного коду, які часто використовуються. Виклик підпрограми здійснюється за допомогою однієї з команд:

call – виконує виклик підпрограми в межах цілої програмної пам’яті. Команда займає у програмі 4 байти (32 біти), 22 біти з яких відведені під адресу, однак, для переважної більшості моделей програмний лічильник займає 16 біт, тому ця команда може адресувати для цих моделей лише до 128 кБайт FLASH. Команда виконується за 4 такти.

rcall – виконує відносний виклик підпрограми, здійснюючи зміщення вперед чи назад у програмній пам’яті на вказане число відносно значення поточної адреси (програмного лічильника). Значення зміщення адреси програмного лічильника може приймати -2047… +2048 у пам’яті програм. Команда виконується за 3 такти.

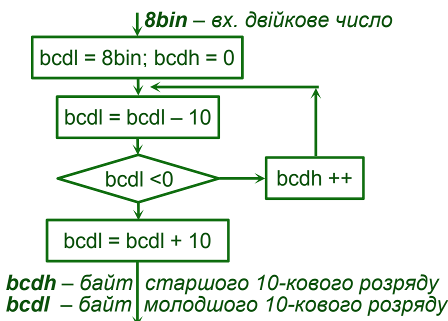
icall – виконує непрямий виклик підпрограми, значення адреси якої зберігається в індексному двобайтному регістрі Z. Об'єм FLASH для цієї команди обмежується 128 кБайтами. Виконується за 3 такти.

При виклику підпрограми за допомогою однієї з команд у стек автоматично заноситься адреса (2 байти) наступної команди після команди виклику. Виконання підпрограми завершується командою **ret** (4 такти), яка присвоює програмному лічильнику значення зворотної адреси, що витягується зі стеку.

Для команд виклику адреси підпрограм вказують за допомогою міток, а компілятор вже сам обчислює потрібні значення.

При використанні підпрограм необхідно обов'язково проініціалізувати стек (!).

```
.include "m32Adef.inc"
.def     _8bin          =r18
.def     _bcdlow        =r18
.def     _bcdhigh       =r19

         .CSEG
         ;Ініціалізація стеку
         ldi     r16, Low(RAMEND)
         out     SPL, r16
         ldi     r16, High(RAMEND)
         out     SPH, r16
         …
         .org    $100
$100:    ldi     _8bin, 35        ;r18 = 35
$101:    rcall   BCD              ;$102 → STACK;   PC=$150
$102:    nop                      ;r19 = 3; r18 = 5
$103:    ldi     _8bin, 71        ;r18 = 71
$104:    rcall   BCD              ;$105 → STACK;   PC=$150
$105:    nop                      ;r19 = 7; r18 = 1
         …
;Підпрограма Двійково-Десяткового Кодування (до 99, незапаковане)
         .org    $150
BCD:     clr     _bcdhigh
BCD1:    subi    _bcdlow, 10
         brcs    BCD2
         inc     _bcdhigh
         rjmp    BCD1
BCD2:    subi    _bcdlow, -10
         ret     ; PC ← STACK
```



3.10. Реалізація переривань.

У МК AVR переривання реалізують механізм оброблення подій від вбудованих периферійний пристроїв. Оскільки моделі AVR відрізняються одна від одної кількістю та різноманітністю вбудованих пристроїв, то відповідно, і кількість переривань у них є різною.

Основною відправною точкою механізму переривань є таблиця векторів переривань. Ця таблиця представляє собою послідовний список адрес у програмній пам’яті (рис. 3.12). Як правило, ця таблиця розміщується на початку адресного простору пам’яті програм. Інформацію про наявні переривання та адреси їхніх векторів переривань можемо отримати в інструкції виробника на конкретну модель.

Address	Labels	Code	Comments
$000	jmp	RESET	; Reset Handler
$002	jmp	EXT_INT0	; IRQ0 Handler
$004	jmp	EXT_INT1	; IRQ1 Handler
$006	jmp	EXT_INT2	; IRQ2 Handler
$008	jmp	TIM2_COMP	; Timer2 Compare Handler
$00A	jmp	TIM2_OVF	; Timer2 Overflow Handler
$00C	jmp	TIM1_CAPT	; Timer1 Capture Handler
$00E	jmp	TIM1_COMPA	; Timer1 CompareA Handler
$010	jmp	TIM1_COMPB	; Timer1 CompareB Handler
$012	jmp	TIM1_OVF	; Timer1 Overflow Handler
$014	jmp	TIM0_COMP	; Timer0 Compare Handler
$016	jmp	TIM0_OVF	; Timer0 Overflow Handler
$018	jmp	SPI_STC	; SPI Transfer Complete Handler
$01A	jmp	USART_RXC	; USART RX Complete Handler
$01C	jmp	USART_UDRE	; UDR Empty Handler
$01E	jmp	USART_TXC	; USART TX Complete Handler
$020	jmp	ADC	; ADC Conversion Complete Handler
$022	jmp	EE_RDY	; EEPROM Ready Handler
$024	jmp	ANA_COMP	; Analog Comparator Handler
$026	jmp	TWI	; Two-wire Serial Interface Handler
$028	jmp	SPM_RDY	; Store Program Memory Ready Handler

Рис. 3.12. Таблиця векторів переривань для ATmega32A

Для того, щоб певна подія для вбудованого пристрою МК могла бути згенерованою, необхідно перш за все активізувати цей периферійний пристрій та дати дозвіл на переривання для цієї події, а також дати загальний дозвіл на переривання у регістрі стану SREG.

При виникненні цієї події виставляється прапорець переривання у відповідному регістрі, і при першій ж можливості, після виконання поточної команди, і якщо немає у цей час іншого переривання, заноситься у стек адреса наступної команди та виконується перехід на адресу вектора переривань для події, при цьому апаратно скидається

прапор переривання. Далі за цієї адресою вектора у таблиці переривань розміщується команда переходу jmp чи rjmp, яка переходить на підпрограму переривання. Виконання підпрограми завершується командою reti, витягується зі стеку адреса повернення та виконується перехід в основну програму. Ці підпрограми, як правило, розміщують одразу ж після таблиці переривань.

У випадку, якщо подія трапилася, виставився для неї відповідний прапор переривання, але загальний дозвіл ще не наданий (в регістрі стану SREG), тоді ця подія буде оброблена при наданні відповідного загального дозволу переривання.

Слід зазначити, що для деяких подій немає відповідних прапорців переривань. Для них переривання генеруються протягом усього часу, допоки присутня відповідна умова, що необхідна для генерації переривання. Відповідно, якщо умови, що викликають переривання, щезнуть до надання дозволів на переривання, то генерації переривання не відбудеться.

Якщо переривання генерується безперестанно, тобто маємо постійну умову для безпрапорцевих переривань чи в черзі стоять переривання з виставленими прапорцями, то між генеруваннями переривань виконується одна команда з основного коду.

Покажемо роботу механізму переривань на прикладі «**зовнішніх переривань**».

Зовнішні переривання це реакція МК на зміну рівня сигналу на його виводах. Такі переривання бувають індивідуальними (INT), тобто для конкретного одного виводу МК, та груповими (PCINT), тобто одне переривання для групи виводів. МК серії tiny мають як мінімум хоча б одне INT0, МК серії mega мають як мінімум два INT0 та INT1. Групові переривання (PCINT) вже залежать від конкретної моделі, і тому можуть бути, а можуть і ні.

Індивідуальні зовнішні переривання (INT) можуть генеруватися при наявності зростаючого чи спадаючого фронту сигналу, або постійного низького рівня на виводі МК (рис. 3.13). При налаштуванню на низький рівень прапори переривання не виставляються та переривання генеруються допоки присутня умова низького рівня.

Групове зовнішнє переривання (PCINT) охоплює, як правило, виводи цілого порту. При цьому ми можемо індивідуально вибирати, які виводи порту будуть генерувати переривання, а які ні. Генерування переривання буде відбуватися при будь-якій зміні рівня сигналу (при зростанні та спаданні фронтів) на будь-якому виводі групи PCINT.

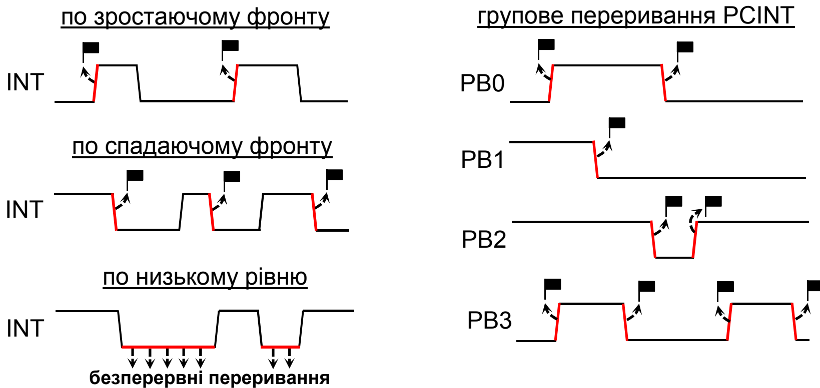


Рис. 3.13. Діаграми генерування зовнішніх переривань

Модель ATmega32A має в наявності 3 зовнішні переривання INT0, INT1 та INT2. INT0 та INT1 можуть бути налаштовані на переривання будь-якої з трьох умов, INT2 – лише на реагування зростаючого чи спадаючого фронту сигналу. Наприклад, у моделі ATmega324P переривання INT2 може бути налаштоване на усі три умови.

Для роботи зі зовнішніми перериваннями в ATmega32A використовуються 4 регістри:

GICR (даються дозволи на переривання INT0, INT1 та INT2);

GIFR (виставляються прапорці переривань);

MCUCR (налаштовуються умови для переривань INT0 та INT1);

MCUCSR (налаштовується умова для переривання INT2).

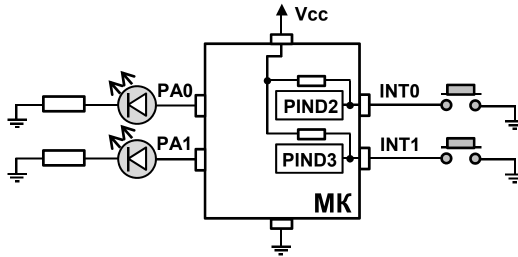
Назви цих регістрів у різних моделей МК можуть бути різними, найчастіше співпадають назви бітів для таких регістрів.

Таблиця 3.10. Умови генерації зовнішніх переривань для ATmega32A

	INT0		INT1		INT2
	ISC01	ISC00	ISC11	ISC10	ISC2
по низькому рівню	0	0	0	0	-
по спадаючому фронту	1	0	1	0	0
по зростаючому фронту	1	1	1	1	1

Продемонструємо роботу зовнішніх переривань на прикладі. Підключимо до виводів INT0 та INT1 кнопки, а самі виводи налаштуємо на вхід з підключеними внутрішніми підтягуючими резисторами. INT0 налаштуємо на переривання по спаду фронту сигналу на виводі, а INT1 на переривання по зростанню фронту сигналу. Переривання

повинні в нас інвертувати сигнали на виходах, відповідно, PA0 та PA1 порту A (світлодіоди будуть засвічуватися/гаснути).




```
.include "m32Adef.inc"
            .CSEG
            .org    $000
            jmp     RESET       ;Reset Handler
            .org    $002
            jmp     EXT_INT0    ;IRQ0 Handler
            .org    $004
            jmp     EXT_INT1    ;IRQ1 Handler

            .org    $028
            reti                ;Store Program Memory Ready Handler
;-------- Підпрограми переривань --------
EXT_INT0:   sbic    PORTA, 0
            rjmp    elsePA0
            sbi     PORTA, 0
            reti
elsePA0:    cbi     PORTA, 0
            reti

EXT_INT1:   sbic    PORTA, 1
            rjmp    elsePA1
            sbi     PORTA, 1
            reti
elsePA1:    cbi     PORTA, 1
            reti
;--------------------------------------------------

            ;ініціалізація стека
RESET:      ldi     r16, Low(RAMEND)
            out     SPL, r16
            ldi     r16, High(RAMEND)
            out     SPH, r16

            ldi     r16, 0x00
            ldi     r17, 0xFF
```

```
        out     DDRA, r17          ;Порт А на вихід
        out     PORTA, r16

        out     DDRD, r16          ;Порт А на вхід
        out     PORTD, r17

        ;Зовнішні переривання INT0, INT1
        ldi     r16, (1<<INT0)|(1<<INT1)      ;вибір переривань
        out     GICR, r16

        ldi     r16, (1<<ISC01) | (0<<ISC00)  |  (1<<ISC11) | (1<<ISC10)
        out     MCUCR, r16  ;INT0-по зрост.рівню; INT1-по спад.фронту

        ;скид прапорців зовнішнього переривання
        ldi     r16, (1<<INTF0) | (1<<INTF1)
        out     GIFR, r16

        ;загальний дозвіл на переривання
        sei
```

```
main:
        rjmp    main
```

У своїй програмі ми не використовуємо усіх наявних для вказаної моделі МК переривань, і тому на початку коду у таблиці переривань ми вписуємо лише вектори рестарту МК та тих переривань, які ми будемо використовувати. Однак правилом хорошого тону є вписувати також останній у таблиці вектор переривання, свого роду як заглушку, що позначає кінець таблиці. Для останнього вектора немає підпрограми, а лише команда виходу з переривання. Це гарантує, що ми не будемо вписувати на місці таблиці переривань свій програмний код.

У таблиці переривань ми можемо використовувати не абсолютну команду переходу jmp, а відносну rjmp, якщо впевнені, що підпрограми переривань знаходяться на віддалі 2 кбайт. Це нам зекономить по 1 такту.

При використанні переривань, обов'язково необхідно проініціалізувати стек, оскільки він використовується для збереження зворотної адреси.

Ще один момент стосується регістру стану SREG. Якщо програма переривання використовує команди, що виставляють прапорці в регістрі стану, тоді перед початком роботи в перериванні необхідно зберегти SREG у стек, а при виході із переривання, відновити його значення зі стеку. Це необхідно зробити, оскільки переривання вклинюються в основну програму, і може трапитися так, що ми командою

порівняння чисел cp виставили прапорці, і хочемо на їхній основі здійснити перехід за допомогою команд Branch, але тут програма пішла на переривання, в якому її підпрограма змінила прапорці у регістрі стану SREG, і при поверненні в основну програму буде вже спотворена логіка для умовного переходу. Тому при перериваннях необхідно зберігати SREG у стеку.

```
Interrupt_subroutine:
        in      r16, SREG        ;рег.заг.призн. ← SREG
        push    r16              ;рег.заг.призн. → стек
        …  програмний код переривання
        …
        pop     r16              ;рег.заг.призн. ← стек
        out     SREG, r16        ;рег.заг.призн. → SREG
        reti                     ;вихід з переривання
```

3.11. Макроси.

Використання асемблерних макросів з першого погляду може здатися схожим на використання підпрограм, що викликаються за допомогою команд call та rcall. Однак, на відміну від підпрограм, вони:

– Не мають окремого адресного розташування за межами основної програми, куди ми переходимо командами переходу. Програмний код макросу підставляється на етапі компіляції у точку виклику. Таким чином, з метою читабельності коду, ми виносимо часто повторювані куски коду в макроси, а під час компіляції вони назад підставляються.

Питання: чому ж тоді не оформити ці куски коду в підпрограми? Переходи на підпрограми займають час, а якщо у нас постійно повторюється 3-4 команди, то тоді немає змісту оформляти це у вигляді підпрограми, і тому легше винести у вигляді макросу. По друге, макроси можуть давати нашій програмі певну гнучкість (про це далі).

Звісно, у порівнянні з підпрограмами, виклики макросів збільшують об'єм програмної пам'яті, тобто, скільки раз викликали макрос, на стільки ж (к-сть викликів * об'єм макросу) і збільшується використання програмної пам'яті. У той час, як підпрограми при багатократному виклику не потребують додаткової пам'яті.

– Макроси, на відміну від підпрограм, можуть зберігатися в окремих файлах. Тобто, ми можемо з легкістю використовувати готові напрацювання, оформлені у вигляді макросів, в інших проектах, чи використовувати у своїх проектах готові «чужі» макроси.

```
.macro  R16mult10                ;r20 = r20*10 = (r20<<1)+(r20<<3)
        lsl     r20
        mov     _temp, r20
        lsl     r20
        lsl     r20
        add     r20, _temp
.endmacro

.def    _temp = r16

        .CSEG
        …
        ldi     r20, 14          ;r20 = 14
        R16mult10                ;r20 = r20*10 = 140
        subi    r20, 121         ;r20 = r20 – 121 = 19
        R16mult10                ;r20 = r20*10 = 190
        subi    r20, 180         ;r20 = r20 – 180 = 10
        R16mult10                ;r20 = r20*10 = 100
        …
```

У наведеному лістингу програми ми проілюстрували використання макросу на прикладі швидкого множення беззнакового однобайтного числа на 10.

– Макроси можуть мати вхідні параметри, що робить їх більш універсальними, аніж використання підпрограм через команди переходу. Як параметри, ми можемо передати у макрос або константи, або назви регістрів. Макрос може приймати до 10 параметрів. Посилання на ці параметри позначаються в середині макросу як @0-@9. Порядок слідування визначається при виклику макросу. Під час компіляції при підстановці коду макросу у точки виклику замість параметризованих змінних підставляються параметри, що вказані через кому після імені викликаного макросу.

```
.include "m32Adef.inc"
.def    _temp = r16

.macro  outi
        ldi     _temp, @1
    .if @0 < 0x40
        out     @0, _temp
    .else
        sts     @0, _temp
    .endif
.endm
```

```
.macro  addi
        ldi     _temp, @1
        add     @0, _temp
.endm

.macro  ldi2
        ldi     _temp, @1
        mov     @0, _temp
.endm

        .CSEG
        outi    DDRA, 0xFF
        outi    PORTA, 0x00

        outi    DDRB, 0x00
        outi    PORTB, 0b01010101

        ldi2    r0, 50          ;r0 = 50
        addi    r0, 25          ;r0 = r0 + 25 = 75

        ldi2    r1, 50          ;r1 = 50
        addi    r1, -25         ;r1 = r1 + -25 = 25
```

DDRA	1	1	1	1	1	1	1	1
PORTA	0	0	0	0	0	0	0	0
DDRB	0	0	0	0	0	0	0	0
PORTB	0	1	0	1	0	1	0	1

У цьому прикладі ми створили 3 макроси, що мають полегшувати процес програмування та робити читабельнішим наш асемблерний код.

Перший макрос outi виводить у будь-який регістр вводу/виводу константу. Зауважте, ми використали у ньому умовну компіляцію. Як ми знаємо, у нафаршированих периферією МК частина регістрів вводу/виводу знаходяться у пам’яті даних після адреси 0x005F, для яких не працюють команди out та in. При роботі з такими регістрами необхідно напряму звертатися до SRAM через відповідні команди доступу до пам’яті. Тому наш макрос спрощує нам задачу запису константи у регістр вводу/виводу, і нам не потрібно слідкувати за тим, в якій області пам’яті знаходиться цей регістр.

Другий макрос addi реалізує команду додавання до будь-якого регістра загального призначення константи.

Третій макрос ldi2 дає можливість присвоювати константу як старшій половині файлу регістрів загального призначення, так і його молодшій частині.

3.12. Робота з даними у SRAM, FLASH та EEPROM.

SRAM. Змінні величини у програмі мовою асемблер ми закріплюємо, як правило, за вільними регістрами загального призначення. Якщо кількість змінних є більшою за кількість регістрів, то ми можемо, наприклад, один регістр використовувати почергово між декількома змінними, і в той час, коли одна змінна використовується з регістром, решта змінних зберігаються у стеку. Цей спосіб потребує ретельного контролю за переміщенням даних у стеку. Тому простіше, при нестачі вільних регістрів загального призначення, розміщувати наші змінні безпосередньо в оперативній пам'яті (SRAM). Звісно, для роботи з нашими змінними в SRAM, необхідно їх буде переміщувати в регістри загального призначення, модифікувати, і назад зберігати в пам'ять даних. Для цього слід передбачити декілька регістрів, що будуть використовуватися як тимчасові змінні.

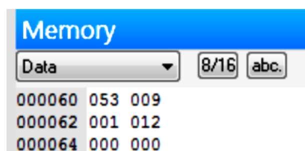
Для зберігання наших даних в SRAM необхідно перш за все зробити розмітку в цій пам'яті під наші змінні за допомогою директиви .byte.

```
        .DSEG
foo:    .byte   1           ;резервує 1 байт для змінної foo
date:   .byte   3           ;0-день, 1-місяць, 2-останні дві цифри року
```

Для запису в SRAM в асемблері AVR передбачено 12 команд. Більшість з них є однотиповими та відрізняються лише використанням різних індексних регістрових пар X, Y, Z.

Найпростішою є команда sts, вона заносить у вказану адресу комірки пам'яті значення з регістра загального призначення.

```
.CSEG
ldi     r16, 53
sts     foo, r16        ; $060 ← 53     запис у SRAM

ldi     r16, 9
sts     date, r16       ; $061 ← 9      запис у SRAM
ldi     r16, 1
sts     date+1, r16     ; $062 ← 1      запис у SRAM
ldi     r16, 12
sts     date+2, r16     ; $063 ← 12     запис у SRAM
```



Для безпосереднього зчитування даних з комірок SRAM є аналогічна команда lds.

```
lds     r16, foo        ; r16=53 ← $060   зчитування зі SRAM
lsl     r16             ; r16=53*2=106
subi    r16, 6          ; r16=106-6=100
sts     foo, r16        ; $060 ← 100      запис у SRAM
```

Решта асемблерні команди працюють зі SRAM через індексні регістрові пари X, Y, Z. Тобто, назви команд однакові: st (для запису) та ld (для зчитування), але аргументи різні: X, Rd; X+, Rd; -X, Rd; Y, Rd; Y+, Rd; -Y, Rd; Z, Rd; Z+, Rd; -Z, Rd (для команд зчитування аргументи розташовані навпаки). При використанні команд st та ld спершу у вибрану регістру пару заноситься адреса комірки пам'яті в SRAM. Далі, у залежності від алгоритму, використовується одна з трьох нотацій для вибраної регістрової пари (звичайна, з плюсом, з мінусом).

Команди зі звичайною нотацією працюють з комірками SRAM через адреси, що визначені в регістрових парах, наприклад, команда st X, Rd заносить значення з регістра загального призначення у комірку SRAM за адресою, що вказана в регістровій парі X.

Команди з плюсом після регістрової пари у нотації зчитують/записують дані з/у SRAM, а потім збільшують значення адреси в регістровій парі на одиницю.

Команди з мінусом перед регістровою парою у нотації спершу зменшують значення адреси в регістровій парі на одиницю, а потім зчитують/записують дані з/у SRAM.

Покажемо на тривіальному прикладі використання цих команд. Задача полягатиме у зчитуванні з порту А довільних 100 вибірок зі збереженням їх у SRAM та виводом їх на порт В у зворотному порядку (останнє збережене значення піде першим на вивід).

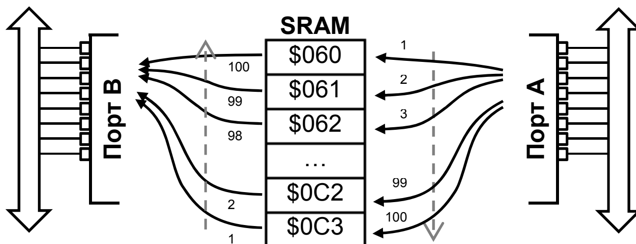


Рис. 3.13. Ілюстрація задачі зворотного вводу-виводу даних

```
.include "m32Adef.inc"
        .DSEG
data:   .byte   100             ; резервуємо 100 байт для наших даних

        .CSEG
        ldi     r16, 0x00
        ldi     r17, 0xFF

        out     DDRA, r16       ; порт A на вхід
        out     PORTA, r16

        out     DDRB, r17       ; порт B на вихід
        out     PORTB, r16
        clr     r20             ; r16=0
        ldi     ZL, low(data)   ; Z ← значення адреси мітки data
        ldi     ZH, high(data)

input:  in      r16, PINA       ; r16 ← порт A
        st      Z+, r16         ; SRAM ← r16;   (Z++)
        inc     r20             ; r16++
        cpi     r20, 100
        brne    input           ; if(r16!=100) перехід на мітку input

        clr     r20             ; r16=0
output: ld      r16, -Z         ; (Z--);    r16 ← SRAM
        out     PORTB, r16      ; порт B ← r16
        inc     r20             ; r16++
        cpi     r20, 100
        brne    output          ; if(r16!=100) перехід на мітку output
```

Також є ще дві корисні команди для роботи зі SRAM: std Y+k, Rr та ldd Rr, Y+k. Ці команди працюють лише із регістровими парами Y та Z, та виконують непрямий відносний запис/зчитування у/з SRAM. Адреса комірки пам'яті, до якої ми звертаємося, отримується в результаті сумування значення регістрової пари та константи, при цьому значення регістрової пари не змінюється.

```
        ldi     ZL, low(date)
        ldi     ZH, high(date)  ; Z=$060 ← адреса мітки data

        ldi     r16, 25         ; r16=25
        std     Z+2, r16        ; $062 ← 25      (Z=$060)
        std     Z+5, r16        ; $065 ← 25      (Z=$060)
        ldd     r16, Z+10       ; r16 ← $06A     (Z=$060)
```

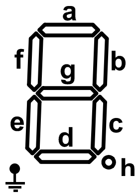
Команди для роботи зі SRAM виконується МК за 2 такти.

FLASH. Для реалізації деяких задач у програмі іноді виникає необхідність мати у наявності таблиці наперед визначених даних, наприклад, таблиця синусів, косинусів, вектори значень для двійково-десяткового кодування тощо. Найпростіше ці значні масиви даних зберігати безпосередньо в пам’яті програм, в області після коду програми, щоб наші таблиці з даними не перешкоджали його виконанню. Звичайно, ми можемо і посеред програми зробити острівок з даними та перестрибувати його за допомогою команд переходу.

Покажемо роботу з пам’яттю програм на конкретному прикладі, використовуючи вивчений вже нами матеріал. Реалізуємо простеньку задачу по циклічному виводу чисел від 1 до 99 з інтервалом в 0,5 сек. на семисегментні індикатори.

Семисегментний графічний індикатор складається зі світлодіодних сегментів. Засвічуючи певні сегменти, можемо вивести необхідне число. Семисегментні індикатори є зі спільним анодом (+) та спільним катодом (–). Це означає, що у такому індикаторі в усіх сегментах один з виводів об’єднаний з рештою подібних виводів інших сегментів. І тоді на спільний вивід подається або високий рівень напруги (+), або низький (–), у залежності від конструкції вибраного індикатора. На решта виводів подають протилежні за рівнем напруги для засвічування сегментів.

Розглянемо індикатор зі спільним катодом. На спільний вивід подаємо низький рівень, а для засвічування сегментів необхідно з МК подавати високі рівні. На рис. 3.14 зображено вигляд індикатора з прийнятою літерною нумерацією сегментів, та наведена таблиця з відповідними кодами, які необхідно подати на індикатор, щоб засвітилася певна цифра. Наприклад, для висвічування цифри «5» необхідно засвітити сегменти a, f, g, c, d, тобто подати на них високий рівень.



Зобр.		h	g	f	e	d	c	b	a	10-знач.	16-знач.
0	0b	0	0	1	1	1	1	1	1	63	3F
1	0b	0	0	0	0	0	1	1	0	6	6
2	0b	0	1	0	1	1	0	1	1	91	5B
3	0b	0	1	0	0	1	1	1	1	79	4F
4	0b	0	1	1	0	0	1	1	0	102	66
5	0b	0	1	1	0	1	1	0	1	109	6D
6	0b	0	1	1	1	1	1	0	1	125	7D
7	0b	0	0	0	0	0	1	1	1	7	7
8	0b	0	1	1	1	1	1	1	1	127	7F
9	0b	0	1	1	0	1	1	1	1	111	6F

Рис. 3.14. Кодування семисегментного індикатора зі спільним катодом

Схему підключення семисегментних індикаторів до портів МК виконаємо, як на рис. 3.15.

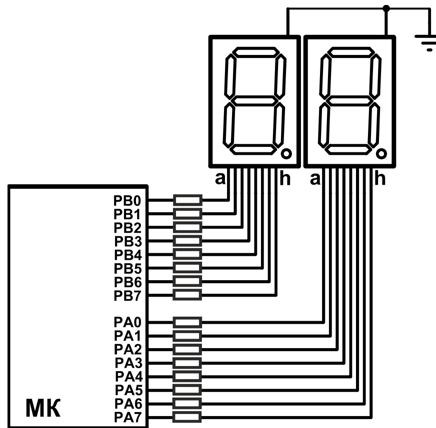


Рис. 3.15. Схема підключення індикаторів для задачі виводу числа

Як ми вже згадували, для розмітки простору у пам'яті FLASH використовуються такі директиви: .db (байт), .dw (слово або 2 байти), .dd (2 слова або 4 байти) та .dq (4 слова або 8 байт). Для нашої задачі нам необхідно створити таблицю з 10-ти однобайтних значень, які б відповідали кодам виводу від 0 до 9 для нашого семисегментного індикатора. Значення розміщуються у таблиці у порядку зростання відображуваного числа. Тобто, якщо нам необхідно буде вивести число «5», тоді ми до адреси мітки таблиці додамо значення зсуву 5 і отримаємо необхідний код.

Зчитування значення з пам'яті FLASH виконується командою lpm (3 такти) згідно адреси, що вказана у регістровій парі Z. Тобто, нам спершу необхідно загрузити в Z адресу мітки таблиці, а потім до Z додати необхідне зміщення у таблиці та виконати команду зчитування.

```
.include "m32Adef.inc"
.def      _8bin    =r18
.def      _bcdL    =r18
.def      _bcdH    =r19
; Константи
.equ      Fig_0 = 63          ; 0
.equ      Fig_1 = 6           ; 1
.equ      Fig_2 = 91          ; 2
.equ      Fig_3 = 79          ; 3
.equ      Fig_4 = 102         ; 4
```

```
.equ      Fig_5 = 109        ; 5
.equ      Fig_6 = 125        ; 6
.equ      Fig_7 = 7          ; 7
.equ      Fig_8 = 127        ; 8
.equ      Fig_9 = 111        ; 9

          .CSEG
          ldi     r16, Low(RAMEND)   ┐
          out     SPL, r16           ├ Ініціалізація стеку
          ldi     r16, High(RAMEND)  │
          out     SPH, r16           ┘

          ldi     r16, 0x00
          ldi     r17, 0xFF
          out     DDRA, r17          ; порт А на вихід
          out     PORTA, r16
          out     DDRB, r17          ; порт В на вихід
          out     PORTB, r16

          clr     r20                ; r20=0  (число для виводу на індикатори)

main:     mov     _8bin, r20         ; r18 ← r20
    ;---------- Виклик підпрограми 2-10 кодування ----------------------------
          rcall   BCD       ; вхід: _8bin;    вихід: _bcdL(мол.),  _bcdH(старш.)

    ;---------- Вивід молодшого розряду десяткового числа ---------------
          ldi     ZL, low(SegTable*2)
          ldi     ZH, high(SegTable*2)    ;Z ← адреса мітки SegTable у байтах
          clr     r16
          add     ZL, _bcdL
          adc     ZH, r16                  ; Z = Z + _bcdL (зміщення)
          lpm     r16, Z                   ; r16 ← FLASH(Z)
          out     PORTA, r16               ; порт А ← r16

    ;---------- Вивід старшого розряду десяткового числа -----------------
          ldi     ZL, low(SegTable*2)
          ldi     ZH, high(SegTable*2)    ;Z ← адреса мітки SegTable у байтах
          clr     r16
          add     ZL, _bcdH
          adc     ZH, r16                  ; Z = Z + _bcdH (зміщення)
          lpm     r16, Z                   ; r16 ← FLASH(Z)
          out     PORTB, r16               ; порт В ← r16

    ;---------- Виклик підпрограми затримки -----------------------------------
          rcall   P05sec
    ;---------- Інкрементування числа r20 до 99 ---------------------------------
          inc     r20                ; r20++
          cpi     r20, 100
          brne    main               ; if (r20 != 100)  goto main
          clr     r20                ; else  r20=0;  goto main
          rjmp    main
```

```
;Підпрограма Двійково-Десяткового Кодування (до 99, незапаковане)
BCD:    clr     _bcdH
BCD1:   subi    _bcdL, 10
        brcs    BCD2
        inc     _bcdH
        rjmp    BCD1
BCD2:   subi    _bcdL, -10
        ret

;Підпрограма паузи 0.5 сек.
P05sec: ldi     r16, 0x00
        ldi     r17, 0x35
        ldi     r18, 0x0C
delay:  subi    r16, 1
        sbci    r17, 0
        sbci    r18, 0
        brne    delay
        ret

;Вектор даних
SegTable: .db   Fig_0,Fig_1,Fig_2,Fig_3,Fig_4,Fig_5,Fig_6,Fig_7,Fig_8,Fig_9
     ;коди цифр  0     1     2     3     4     5     6     7     8     9
```

Частини коду у програмі, що мають відношення до роботи з пам'яттю FLASH, відмічені заокругленими прямокутниками:

– розміщення таблиці даних у пам'яті програм з використанням директиви .db ;
– читання необхідного значення з пам'яті за адресою, що розміщена в регістровій парі Z.

EEPROM. Ця пам'ять, аналогічно SRAM, розташована у своєму адресному просторі та організована лінійно. Доступ до пам'яті побайтний. Для роботи з EEPROM використовуються 3 регістри вводу/виводу:

EEAR – регістр адреси EEPROM-пам'яті, який фізично розташовується у двох регістрах EEARH: EEARL. У цей регістр завантажується адреса комірка, до якої буде звертатися звертання, як для запису, так і для читання.

EEDR – регістр даних. У нього завантажуються дані для запису в EEPROM, а також у ньому розміщуються дані, отримані при читанні з EEPROM-пам'яті.

EECR – регістр керування доступом до EEPROM-пам'яті. У ньому задіяні 4 біти для визначення поведінки МК при роботі з EEPROM.

3	EERIE	Дозвіл на переривання по завершенню запису в EEPROM
2	EEMWE	Попередній дозвіл на запис даних. Після нього повинен одразу бути встановлений біт EEWE, інакше він буде апаратно скинутий через 4 такти.
1	EEWE	Дозвіл на запис. При встановленні в 1 відбувається запис даних у EEPROM (при умові, що встановлений біт EEMWE).
0	EERE	Дозвіл на читання**.** При встановленні в 1 відбувається читання з EEPROM. По завершенню читання цей розряд скидається апаратно.

Алгоритм запису одного байту в EEPROM-пам'ять:

1. Дочекатися готовності EEPROM (поки не скинеться біт EEWE).
2. Завантажити байт даних у регістр EEDR, а необхідну адресу у регістр EEAR.
3. Встановити в «1» біт EEMWE регістра керування EECR.
4. Записати у розряд EEWE регістра керування логічну «1» протягом 4-х тактів.

Тривалість запису в EEPROM є дуже великою та складає приблизно 2-9 мсек. Тому послідовний запис деякого числа байтів в EEPROM можемо організувати через проміжний буфер FIFO. Тобто, ми запишемо дані усі підряд в буфер, а він, використовуючи процедуру переривання по завершенню запису, буде по байту записувати наші дані в EEPROM, працюючи при цьому у фоновому режимі. Про організацію таких буферів ми розкажемо при розгляді роботи послідовного порту USART.

Ще один необхідно звернути увагу на те, що якщо між пп.3 та 4 відбудеться переривання, то зірветься уся процедура запису. Тому необхідно на цей час забороняти будь-які переривання.

Процедура читання має подібний алгоритм: очікуємо готовності EEPROM, далі завантажуємо необхідну адресу у регістр EEAR, встановлюємо в «1» біт дозволу на читання EERE та зчитуємо наш байт даних з регістра EEDR. Читання відбувається за 1 такт, однак прогальмовування у 4 такти відбувається при встановленні біта дозволу на читання EERE.

Наведемо приклад програми для роботи з EEPROM. Записане у EEPROM, а потім зчитане назад з нього число виводимо на лінійку зі світлодіодів, що підключені до порту А (рис. 3.16).

```
.include "m32Adef.inc"
        .CSEG
        ldi     r16, 0x00
        ldi     r17, 0xFF
        out     DDRA, r17               ; порт A на вихід
        out     PORTA, r16
        ldi     r18, 159                ; r18=0b10011111
;---------- запис в E_Pass ----------------------------
E_Write:sbic    EECR, EEWE              ; очікуємо готовності EEPROM
        rjmp    E_Write
        ldi     r17, high(E_Pass)       ; отримуємо адресу мітки E_Pass
        ldi     r16, low(E_Pass)
        out     EEARH, r17              ; завантажуємо адресу в рег. EEAR
        out     EEARL, r16
        out     EEDR, r18               ; заносимо байт даних для запису
        cli
        sbi     EECR, EEMWE             ; вст. попередній дозвіл на запис
        sbi     EECR, EEWE              ; вст. остаточний дозвіл на запис
        sei
;---------- читання з E_Pass ----------------------------
E_Read:sbic     EECR, EEWE              ; очікуємо готовності EEPROM
        rjmp    E_Read
        ldi     r17, high(E_Pass)       ; отримуємо адресу мітки E_Pass
        ldi     r16, low(E_Pass)
        out     EEARH, r17              ; завантажуємо адресу в рег. EEAR
        out     EEARL, r16
        sbi     EECR, EERE              ; вст. дозвіл на читання
        in      r20, EEDR               ; зчитуємо байт з рег. EEDR
        out     PORTA, r20              ; виводимо байт на світлодіоди

main:   rjmp    main

;розмітка памяті EEPROM
        .ESEG
E_Pass: .db     145     ;0b10010001
```

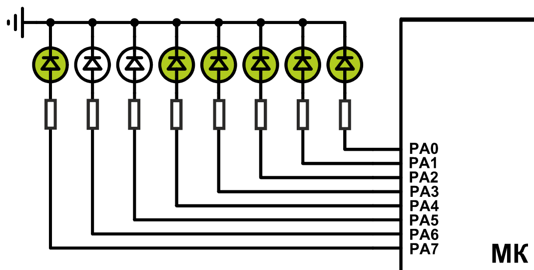


Рис. 3.16. Результат виконання програми при роботі з EEPROM

Розмітку областей EEPROM-пам’яті можемо виконувати, як у пам’яті програм, за допомогою директив .db, .dw, .dd, .dq, що вказують на розміщення даних, так і за допомогою директиви .byte, що резервує необхідну кількість байтів (як у SRAM).

3.13. Таймери.

МК AVR мають на борту різні реалізації таймерів. Перш за все це 8-ми та 16-ти розрядні таймери/лічильники. Останні, відповідно, можуть відраховувати точніші та довші інтервали часу. Також присутні в моделях AVR і сторожові таймери (Watchdog Timer), що призначені для контролю за несанкціонованим зациклюванням програми.

8-розрядні таймери/лічильники. У МК AVR реалізовано три виконання 8-розрядних таймерів/лічильників, що відрізняються набором виконуваних функцій. Перше виконання має реалізовані лише восьмирозрядний лічильник та лічильник зовнішніх подій. Друге виконання має додатково 8-розрядний ШІМ (широтно-імпульсний генератор) та формувач сигналів. Третє виконання ще додає керування таймером/лічильником в асинхронному режимі (годинник реального часу). У моделях AVR можуть бути присутні 2 восьмирозрядні таймери, які позначаються відповідно T0 та T2. Кількість таймерів та їхнє виконання залежать від конкретної моделі. Також від виконання залежить і кількість переривань, що можуть бути згенеровані таймером.

Рис. 3.17. Регістри вводу/виводу 8-розрядного таймера T0 ATmega32A

Як 8- так і 16-розрядні таймери AVR використовують у своїй роботі 10-розрядний **попередній подільник частоти** (рис. 3.17). Він використовується для всіх наявних таймерів в моделі МК, окрім таймера з керуванням в асинхронному режимі, який має свій окремий попередній подільник.

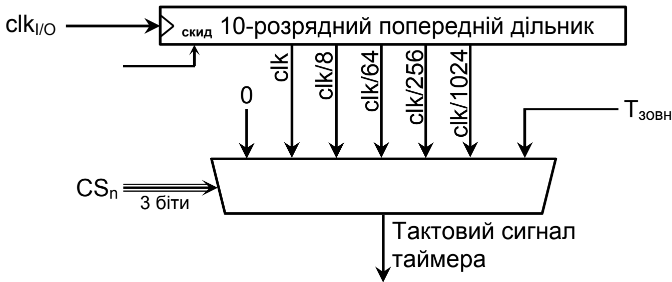


Рис. 3.18. Блок попереднього дільника таймера

На виході мультиплексора дільника частоти вхідна тактова частота, від якої тактується МК, розбита на ряд підчастот, які зменшені у вказане число раз. Наприклад, clk/64 означає, що відраховується 64 тактів МК, після чого посилається імпульс для таймера. Таким чином, ми можемо сповільнити роботу нашого таймера.

Варто мати на увазі, що попередній подільник працює незалежно від таймера. У результаті чого виникає невизначений проміжок часу (від 1 до значення дільника) між дозволом таймера та його першим відліком при сумісній роботі з дільником частоти. Для цього передбачена можливість скиду значень попереднього дільника частоти шляхом запису лог. 1 у відповідний біт регістра SFIOR.

Таймери, що не мають асинхронного режиму роботи, можуть тактуватися також від зовнішнього сигналу $T_{зовн}$. При цьому можна налаштувати відлік або по наростанню фронту сигналу, або по його спаду. Частота зовнішнього сигналу повинна бути у 2,5 рази меншою за частоту тактового генератора МК.

Попередній подільник частоти для асинхронного таймера може тактуватися як від основного генератора тактової частоти МК, так і від автономного генератора із своїм зовнішнім кварцем. Цей автономний генератор оптимізований під годинниковий кварц 32 768 Гц. Асинхронний подільник має додаткові коефіцієнти поділу 32 та 128, однак відсутній підрахунок від зовнішніх імпульсів.

Вибір режиму тактування таймера здійснюється за допомогою трьох бітів CS_n, що розміщені у регістрі $TCCR_n$ (табл. 3.11).

Таблиця 3.11. Вибір тактового сигналу для таймерів виконання 1, 2

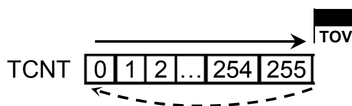
CS_n2	CS_n1	CS_n0	Джерело тактового сигналу
0	0	0	Таймер зупинений
0	0	1	$clk_{I/O}$
0	1	0	$clk_{I/O}/8$
0	1	1	$clk_{I/O}/64$
1	0	0	$clk_{I/O}/256$
1	0	1	$clk_{I/O}/1024$
1	1	0	Рахунок імпульсів на виводі T_n за спадаючим фронтом
1	1	1	Рахунок імпульсів на виводі T_n за наростаючим фронтом

Таймери/лічильники можуть працювати у **4-х режимах**:

– Нормальний режим. 8-розрядний лічильний регістр нараховує 256 тактів таймера, виникає переповнення, і рахунок продовжується зі значення 0. При виникненні переповнення виставляється у регістрі TIFR прапор переповнення TOV та генерується переривання (якщо це переривання дозволене у регістрі TIMSK прапорцем TOIE, а також виставлений загальний дозвіл у регістрі стану SREG).
– Скид при співпадінні. У цьому режимі наш лічильний регістр здійснює рахунок тактів таймера до вказаного 8-розрядного значення у регістрі порівняння OCR, після чого скидається у значення 0. Під час обнулення лічильного регістра виставляється прапор співпадіння OCF та генерується переривання, якщо звісно є загальний дозвіл на переривання у SREG та виставлений прапор дозволу OCIE у регістрі TIMSK.
– Швидкий ШІМ.
– ШІМ з точною фазою.

Деталі останніх 2-х режимів будуть розкриті пізніше.

Переривання по переповненню



Переривання по співпадінню

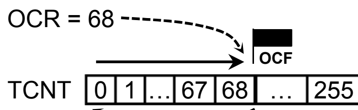


Рис. 3.19. Переривання таймера за значенням лічильника

Таблиця 3.12. Режими роботи 8-розрядних таймерів

WGM_n1	WGM_n0	Режими роботи таймера
0	0	Нормальний (по переповненню)
0	1	ШІМ з точною фазою
1	0	Скид при співпадінні
1	1	Швидкий ШІМ

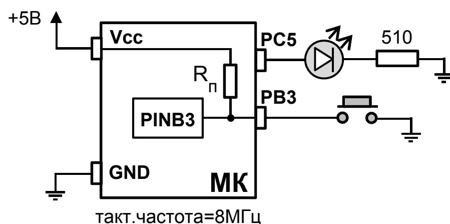
Для нормального режиму та скиду при співпадінні можемо налаштувати поведінку виводу таймера OC_n (табл. 3.13).

Таблиця 3.13. Керування виводом OC у норм. реж. та скиду при співп.

COM_n1	COM_n0	Поведінка
0	0	Таймер відключений від виводу OC
0	1	Стан виводу змінюється на протилежний
1	0	Вивід скидується в «0»
1	1	Вивід скидується в «1»

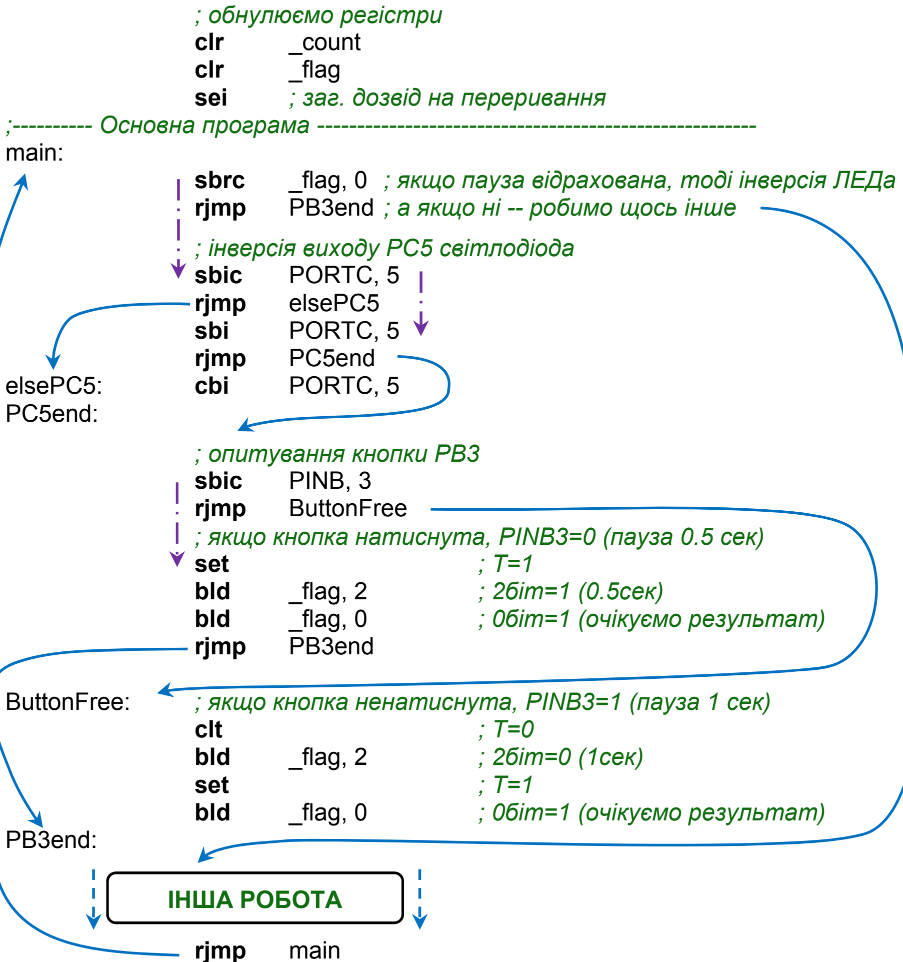
При необхідності стан виводу OC_n може бути змінений примусово, шляхом запису лог. 1 у розряд FOC_n регістра $TCCR_n$. Переривання при цьому не генерується.

Приклад 1. Використовуючи таймер 0, реалізувати блимаючий світлодіод. У пасивному режимі світлодіод блимає з інтервалом приблизно 1 сек., тобто 1 сек. світиться, а на 1 сек. гасне. При натисненні кнопки, світлодіод починає блимати у 2 рази частіше.



При максимальному значенні коефіцієнта попереднього подільника частоти рівному 1024 та тактовій частоті 8 МГц, таймер 0 може відраховувати максимальні інтервали часу 32.768 мсек. Тому для реалізації заданих інтервалів (1 та 0,5 сек) необхідно буде використовувати певне число послідовних відліків часу таймера 0. Для точності інтервалів налаштуємо наш таймер 0 у режим «скид при співпадінні» на час 25 мсек.

```
.include "m32Adef.inc"
.def      _flag  = r0          ; 0біт - пауза =0 (пораховано), =1(ще рахує)
                               ; 2біт - тривалість паузи =0(1сек), =1(0.5сек)
.def      _count = r21
                .CSEG
                .org    $000
                jmp     RESET          ; Reset Handler
                .org    $014
                jmp     TIM0_COMP      ; Timer0 Compare Handler
                .org    $028
                reti                   ;Store Program Memory Ready Handler
;підпрограма переривання по співпадінню Таймера 0 ---------------
TIM0_COMP:      sbrs    _flag, 0       ; якщо відлік дозволений
                rjmp    Task0end
                inc     _count         ; _count++
                sbrs    _flag, 2       ; якщо 2біт=1, тоді рахуємо до 20
                rjmp    sec1
                cpi     _count, 20     ; порівнюємо з числом 20
                brne    Task0end       ; на кінець, якщо не рівне
                rjmp    ready
sec1:           cpi     _count, 40     ; порівнюємо з числом 40
                brne    Task0end       ; на кінець, якщо не рівне
ready:          clt                    ; T=0
                bld     _flag, 0       ; 0біт=1 (пауза порахована)
                clr     _count         ; _count=0
Task0end:
T0end:          reti

;-------------------------------------------------------------------------------------
RESET:          ldi     r16, Low(RAMEND)   ┐
                out     SPL, r16           │ Ініціалізація стеку
                ldi     r16, High(RAMEND)  │
                out     SPH, r16           ┘
                ldi     r16, 0x00
                ldi     r17, 0xFF
                out     DDRB, r16      ; порт В на вхід
                out     PORTB, r17
                out     DDRC, r17      ; порт С на вихід
                out     PORTC, r16
; таймер_0 скид по співпадінню, 25msec, Prescaler=1024, OCR=0xC2
                ldi     r16, (1<<WGM01)|(1<<CS02)|(1<<CS00)
                out     TCCR0, r16     ; OCR=0xC2
                ldi     r16, 0xC2
                out     OCR0, r16      ; OCR=0xC2
                ldi     r16, (1<<OCIE0)
                out     TIMSK, r16     ; дозвіл на перер. по співпадінню
```

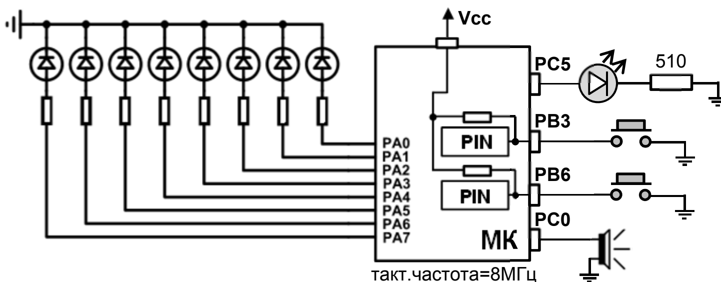


Опис програми. Використання таймера дає нам можливість одночасно відраховувати потрібні інтервали часу та виконувати іншу додаткову роботу. Програмна логіка для відліку пауз між блиманнями світлодіода реалізована на логічних прапорцях (окремих бітах) регістра загального призначення R0 (псевдонім _flag). На початку основної програми здійснюємо перевірку нульового біта R0. Якщо він скинутий (=0), тоді здійснюємо інверсію виводу МК, до якого підключений світлодіод. Далі опитуємо ногу МК, до якої підключена кнопка. Якщо кнопка не натиснута, тоді на вході логічна «1» (на початку програми ми ініціалізували цей порт на вхід з підтягуючими

резисторами). Далі скидаємо 2 біт R0 в логічний «0» (вказівка для таймера відрахувати 1 сек), та встановлюємо в логічну «1» нульовий біт (очікування).

Задані інтервали часу відраховуються у підпрограмі переривання таймера 0 по співпадінню. Співпадіння налаштовані на час 25 мсек. Відповідно, для інтервалу в 1 сек. необхідно відрахувати 40 таких переривань таймера 0, а для 0,5 сек. – 20 переривань. Відлік кількості переривань ведеться у регістрі R21 (псевдонім _count), і на кожному перериванні виконується порівняння із заданою константою.

Приклад 2. Для попереднього прикладу додамо ще одну задачу, яка буде працювати паралельно з основною. Для цього приєднаємо ще одну кнопку PB6, а на порт А вісім світлодіодів. При натиску кнопки регістр порту А збільшуватиметься на 1. Значення регістра буде відображатися у бінарному вигляді на лінійці світлодіодів. При кожному натиску кнопки звуковипромінювач, під'єднаний до виводу PC0, має видавати короткий сигнал.



Задачу підрахунку кількості натискань кнопки розмістимо у програмний блок «інша робота» попереднього програмного лістингу, а також додамо невеликі фрагменти коду у сегменти макровизначень, ініціалізації та у підпрограму переривань таймера 0.

<u>У сегменті з макровизначеннями додамо:</u>

```
.def    _flag2  = r1    ; 0біт - пауза =0 (пораховано), =1(ще рахує)
                        ; 2біт - стан кнопки PB6 =0(відпущена), =1(натиснута)
                        ; 4біт - звуковипромінювач =0(тихо), =1(пищить)
.def    _count2= r22
.def    _countA= r23
.def    _countBuz = r24
```

У блок ініціалізації периферії:

```
        out     DDRA, r17
        out     PORTA, r16
        clr     _count2
        clr     _countA
        clr     _countBuz
        clr     _flag2
```

У сегменті основної програми у блоці «інша робота»:

```
; --------- Робота №2 ---------------------------------------------------------------------------
        sbrc    _flag2, 0       ;якщо паузу порахов., тоді заг. робота
        rjmp    PB6end          ;а якщо ні -- робимо щось інше
        sbrs    _flag2, 2       ;Якщо кнопка PB6 була натиснута
        rjmp    Label1
        sbis    PINB, 6         ;перевіряємо чи кнопка PB6 відпущена
        rjmp    PB6end
        clt                     ; T=0
        bld     _flag2, 2       ;кнопка PB6 відпущена
        set                     ; T=1
        bld     _flag2, 0       ;на паузу від брязкоту контактів
        rjmp    PB6end
Label1: sbic    PINB, 6
        rjmp    PB6end
        set                     ; T=1
        bld     _flag2, 2       ;кнопка PB6 натиснута
        bld     _flag2, 0       ;на паузу від брязкоту контактів
        bld     _flag2, 4       ;звук
        sbi     PORTC, 0
        inc     _countA
        out     PORTA, _countA
PB6end:
```

У кінець підпрограми таймера перед командою виходу reti додамо такий код:

```
;---------------------Task2 (затримка)--------------------
        sbrs    _flag2, 0
        rjmp    Task2end
        inc     _count2
        cpi     _count2, 4
        brne    Task2end
        clt
        bld     _flag2, 0       ;(пораховано)
        clr     _count2         ;(обнулення)
Task2end:
```

```
;--------------------Task3 (звук)------------------------
                    sbrs    _flag2, 4
                    rjmp    Task3end
                    inc     _countBuz
                    cpi     _countBuz, 6
                    brne    Task3end

                    clt
                    bld     _flag2, 4           ;(пораховано)
                    clr     _countBuz           ;(обнулення)
                    cbi     PORTC, 0            ;викл. звук
Task3end:
```

Таким чином, наші задачі можуть паралельно працювати незалежно одна від одної, відраховуючи необхідні інтервали часу та використовуючи при цьому лише один таймер.

3.14. Використання асинхронних таймерів.

МК сімейства Mega у більшості випадків також мають в себе на борту і таймер/лічильник третього виконання, що може працювати в асинхронному режимі, тобто у якості годинника реального часу. Тактовий генератор такого таймера може працювати від окремого зовнішнього кварцу, підключеного до виводів TOSC1 та TOSC1, або від зовнішнього сигналу, що подається на вивід TOSC1. Хоча тактовий генератор таймера і налаштований на частоту 32 768 Гц, але частота кварцового резонатора чи зовнішнього сигналу може знаходитися у межах 0…256 кГц.

Для роботи таймера в асинхронному режимі призначений регістр ASSR. Для переведення таймера в асинхронний режим необхідно встановити біт AS_n в 1. Регістр ASSR має ще три біти, що призначені для контролю стану обновлення регістрів таймера при переключенні в асинхронний режим. У табл. 3.14 наведено режими тактування асинхронного таймера. Решта налаштувань такі ж, як у звичайних 8-розрядних таймерах.

При початковій ініціалізації таймера в асинхронному режимі слід дотримуватися рекомендованої виробником процедури:

- витримати 1 секунду паузи для запуску тактового генератора таймера/лічильника;
- заборонити переривання від таймера/лічильника;
- переключити його в асинхронний режим;
- записати необхідні значення в регістри $TCNT_n$, OCR_n і $TCCR_n$.

- очікувати, поки не будуть скинуті прапорці TCN_nUB, OCR_nUB, TCR_nUB регістру ASSR.
- скинути прапорці переривань таймера/лічильника;
- дозволити переривання.

Таблиця 3.14. Вибір тактового сигналу для таймерів виконання 3

CS_n2	CS_n1	CS_n0	Джерело тактового сигналу
0	0	0	Таймер зупинений
0	0	1	clk
0	1	0	clk/8
0	1	1	clk/32
1	0	0	clk/64
1	0	1	clk/128
1	1	0	clk/256
1	1	1	clk/1024

* *Вибір джерела тактування залежить від значення біта AS_n*

Приклад. Для моделі ATmega32 на основі асинхронного таймера T2 з кварцовим резонатором 32 768 Гц реалізувати секундомір з відліком секунд та хвилин, та їхнім відображенням на 4-х семисегментних елементах зі спільним анодом.

Відлік часу секундоміра реалізується згідно блок-схеми, зображеної на рис. 3.20.

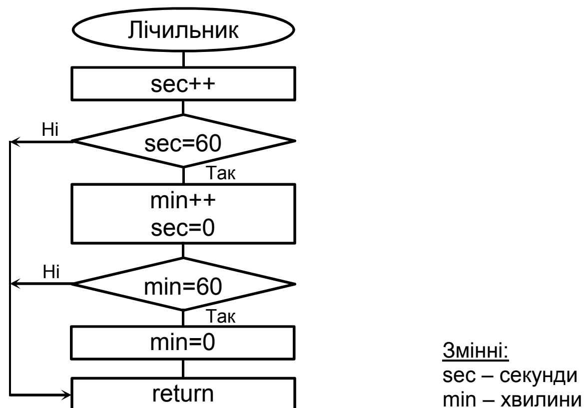


Рис. 3.20. Блок-схема підпрограми відліку часу

На рис. 3.21 зображена принципова схема для нашого прикладу. Для керування напругою на спільних анодах семисегментних елементів використовуються ключі на основі pnp-транзисторів. Для спрощення симуляції в пакеті Proteus ці ключі замінюємо на логічні інвертори.

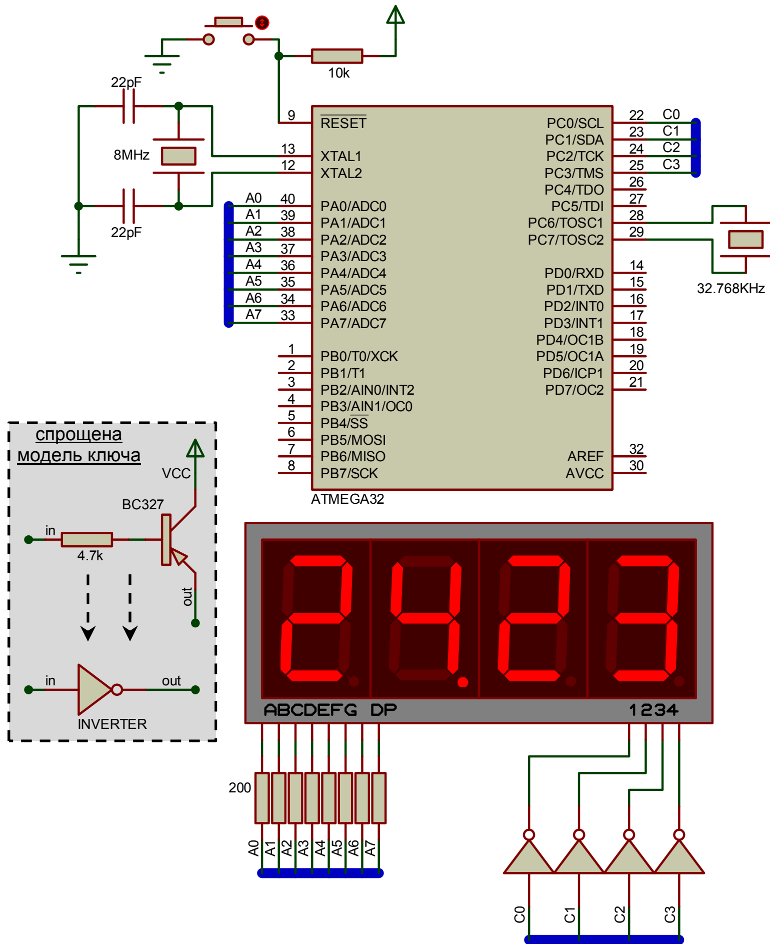
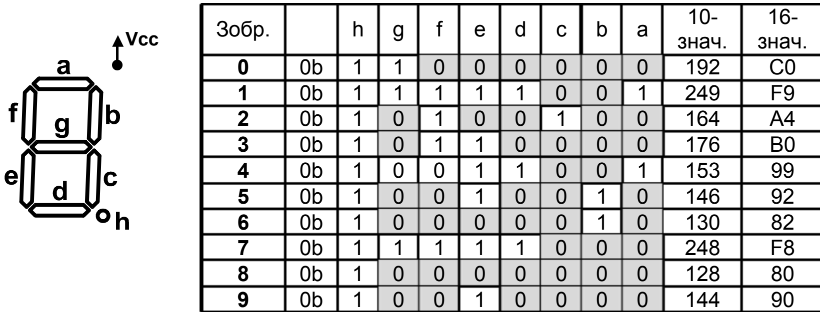


Рис. 3.21. Принципова схема секундоміра

Для відображення чисел на нашому 4-х елементному семисегментнику ми задіяли порт A для засвічування окремих сегментів та половину порту C для керування ключами при подачі напруги живлення на елементи. Наш дисплей працює в режимі динамічної індикації, суть якої полягає у тому, що семисегментні елементи

засвічуються по черзі, тобто в конкретний момент часу світиться лише один семисегментний елемент. Для ефекту постійного свічення, не відчутного для людського ока, кожен елемент повинен засвічуватися з частотою не нижче 60-70 Гц. Оскільки у нас 4 елементи, то загальна частота перемикання ключів подачі живлення на семисегментні елементи повинна складати близько 70×4=280 Гц. Задачу перемикання ключів живлення та подачу байта числа, що має засвічуватися, для кожного семисегментного елемента реалізуємо у підпрограмі переривання таймера T0.

Для відображення чисел на семисегментних елементах потрібно скласти таблицю відповідностей, аналогічну табл. 3.14. Для подачі напруги живлення Vcc на семисегментний елемент слід подати на ключ (pnp-транзистор) низький рівень (лог. «0») з відповідного виводу порту C. Далі, для засвічування сегментів, на їхні виводи катодів через обмежувальні резистори подаються низькі рівні з виводів порту A.



Зобр.		h	g	f	e	d	c	b	a	10-знач.	16-знач.
0	0b	1	1	0	0	0	0	0	0	192	C0
1	0b	1	1	1	1	1	0	0	1	249	F9
2	0b	1	0	1	0	0	1	0	0	164	A4
3	0b	1	0	1	1	0	0	0	0	176	B0
4	0b	1	0	0	1	1	0	0	1	153	99
5	0b	1	0	0	1	0	0	1	0	146	92
6	0b	1	0	0	0	0	0	1	0	130	82
7	0b	1	1	1	1	1	0	0	0	248	F8
8	0b	1	0	0	0	0	0	0	0	128	80
9	0b	1	0	0	1	0	0	0	0	144	90

Рис. 3.22. Кодування семисегментного індикатора зі спільним анодом

Розрахуємо значення для ініціалізації наших таймерів.

T2 (асинхр. таймер) для відліку інтервалів часу 1 сек.:

$$f_{\text{сигн.T2}} = \frac{f_{\text{такт.генер.T2}}}{256 \cdot K_{\text{подільн.T2}}}$$

$$K_{\text{подільн.T2}} = \frac{f_{\text{такт.генер.T2}}}{256 \cdot f_{\text{сигн.T2}}} = \frac{32\ 768\ \text{Гц}}{256 \cdot 1\text{Гц}} = 128$$

T0 для задання частоти переключення ключів живлення:

$$f_{\text{сигн.T0}} = \frac{f_{\text{такт.генер.T0}}}{(OCR0 + 1) \cdot K_{\text{подільн.T0}}}$$

$$\text{OCR0} = \frac{f_{\text{такт.генер.T0}}}{K_{\text{подільн.T0}} \cdot f_{\text{сигн.T0}}} - 1 = \frac{8\ \text{МГц}}{1024 \cdot 280\text{Гц}} - 1 = \sim 27 = 0\text{x1B}$$

```
.include "m32def.inc"
;Імена для регістрів загального призначення-------------------------------------
.def      _temp1      =r16
.def      _temp2      =r17
.def      _temp3      =r18
.def      _segment    =r18

.def      _sec        =r20
.def      _min        =r21
.def      _secL       =r22
.def      _secH       =r23
.def      _minL       =r24
.def      _minH       =r25

;Константи для сегментів дисплею-----------------------------------------------
.equu     Fig_0 = 0xC0      ; 0
.equ      Fig_1 = 0xF9      ; 1
.equ      Fig_2 = 0xA4      ; 2
.equ      Fig_3 = 0xB0      ; 3
.equ      Fig_4 = 0x99      ; 4
.equ      Fig_5 = 0x92      ; 5
.equ      Fig_6 = 0x82      ; 6
.equ      Fig_7 = 0xF8      ; 7
.equ      Fig_8 = 0x80      ; 8
.equ      Fig_9 = 0x90      ; 9

;Підпрограма Двійково-Десяткового Кодування (до 99, незапаковане)
.macro    BCD        ;@0-Bin, @1-BCDL, @2-BCDH
                     mov     @1, @0              ; BCDL <- Bin
                     clr     @2
BCD_1:               subi    @1, 10              ; BCDL =  BCDL - 10
                     brcs    BCD_2               ; якщо BCDL < 0
                     inc     @2                  ; BCDH++
                     rjmp    BCD_1
BCD_2:               subi    @1, -10             ; BCDL =  BCDL + 10

                     clr     _temp1
                     ldi     ZL, low(SegTable*2)   ; Z ← адреса мітки SegTable
                     ldi     ZH, high(SegTable*2)  ;        у байтах
                     add     ZL, @1
                     adc     ZH, _temp1          ; Z = Z + BCDL (зміщення)
                     lpm     @1, Z               ; BCDL ← FLASH(Z)

                     ldi     ZL, low(SegTable*2)   ; Z ← адреса мітки SegTable
                     ldi     ZH, high(SegTable*2)  ;        у байтах
                     add     ZL, @2
                     adc     ZH, _temp1          ; Z = Z + BCDH (зміщення)
                     lpm     @2, Z               ; BCDH ← FLASH(Z)
.endmacro
```

```
            .CSEG
            .org    $000
            jmp     reset
            .org    $00A
            jmp     TIM2_OVF        ; Timer2 Overflow Handler
            .org    $014
            jmp     TIM0_COMP       ; Timer0 Compare Handler

;Переривання при співпадінні таймера T0
TIM0_COMP:  ldi     _temp1, 0xFF
            eor     _temp1, _segment        ; інверсія бітів segment
            out     PORTC, _temp1           ; вмикаємо один з ключів
            sbrc    _segment, 0             ; якщо segment.bit0<>0
            out     PORTA, _minH            ; тоді PORTA←ст. хвилини
            sbrc    _segment, 1             ; якщо segment.bit1<>0
            out     PORTA, _minL            ; тоді PORTA←мол. хвилини
            sbrc    _segment, 2             ; якщо segment.bit2<>0
            out     PORTA, _secH            ; тоді PORTA←ст. секунди
            sbrc    _segment, 3             ; якщо segment.bit3<>0
            out     PORTA, _secL            ; тоді PORTA←мол. секунди

            lsl     _segment                ; segment << 1
            cpi     _segment, (0b10000)
            brne    endCOMP                 ; якщо segment = 0b10000
            ldi     _segment,1              ; тоді segment = 0b00001
endCOMP:    reti

; Переривання по переповненню таймера T2
TIM2_OVF:   inc     _sec                    ; sec++
            cpi     _sec, 60
            brne    endOVF                  ; якщо sec <> 60
            inc     _min                    ; min++
            clr     _sec                    ; sec=0
            cpi     _min, 60
            brne    endOVF                  ; якщо min <> 60
            clr     _min                    ; min=0
endOVF:     BCD     _sec, _secL, _secH      ; розбивка сек. на 2 числа
            BCD     _min, _minL, _minH      ; розбивка хвил. на 2 числа
            andi    _minL, 0b01111111       ; засвічуємо роздільну точку
                                            ; між хвил. та сек.
            reti
;-----------------------------------------------------------------
reset:
            ;ініціалізація стека
            ldi     _temp1, Low(RAMEND)
            out     SPL, _temp1
            ldi     _temp1, High(RAMEND)
            out     SPH, _temp1
;Очікуємо стабілізації зовнішнього кварца 32.768 кГц ~1секунду
            call    delay1sec
```

```
                ;ініціалізація асинхронного таймера
                ldi     _temp1, 0x00
                out     TIMSK, _temp1   ;заборона переривань таймерів
                ldi     _temp1, (1<<AS2)
                out     ASSR, _temp1    ;перехід в асинхронний режим
                ldi     _temp1, (1<<CS22)|(1<<CS20)
                out     TCCR2, _temp1   ;поділ=128; по переповненню

loop:           in      _temp1, ASSR
                andi    _temp1, (1<<TCN2UB)|(1<<OCR2UB)|(1<<TCR2UB)
                brne    loop    ;очікуємо готовності асинхр. таймера T2

                ldi     _temp1, 0x00            ;скидаємо прапорці
                out     TIFR, _temp1            ;переривань таймера

                ;ініціалізація таймера T0 (~280Гц)
                ldi     _temp1, (1<<WGM01)|(1<<CS02)|(1<<CS00)
                out     TCCR0, _temp1   ;скид при співпад., подільн=1024
                ldi     _temp1, 0x1B
                out     OCR0, _temp1    ; OCR0=0x1B

                ;ініціалізація портів вводу/виводу
                ldi     _temp1, 0x00
                ldi     _temp2, 0xFF
                ;Порт A -- сегменти (катод)
                out     DDRA, _temp2            ;порт працює на вихід
                out     PORTA, _temp2       ;+5V (семисегм. не світять)
                ;Порт C -- керування живленням (анод)
                out     DDRC, _temp2            ;порт працює на вихід
                out     PORTC, _temp2       ;+5V  (транзист. ключ вимкнений)

                clr     _sec                    ; sec=0
                clr     _min                    ; min=0
                BCD     _sec, _secL, _secH      ; розбивка сек. на 2 числа
                BCD     _min, _minL, _minH      ; розбивка хвил. на 2 числа
                ldi     _segment, 1
                ;дозвіл на переривання по переповн. T2 та співпадінню T0
                ldi     _temp1, (1<<TOIE2)|(1<<OCIE0)
                out     TIMSK, _temp1

                sei     ;загальний дозвіл на переривання
;основний програмний цикл
main:
```

```
                rjmp    main

;Вектор даних
SegTable:  .db  Fig_0, Fig_1, Fig_2, Fig_3, Fig_4, Fig_5, Fig_6, Fig_7, Fig_8, Fig_9
        ;цифри  0      1      2      3      4      5      6      7      8      9
```

```
;Підпрограма затримки 1 секунда
delay1sec:      ldi     _temp1, 0x00
                ldi     _temp2, 0x6A
                ldi     _temp3, 0x18
sec1:           subi    _temp1, 1
                sbci    _temp2, 0
                sbci    _temp3, 0
                brne    sec1
                ret
```

3.15. ШІМ-модуляція.

Широтно-імпульсна модуляція (PWM – Pulse Width Modulation) представляє собою спосіб задання аналогового сигналу цифровим методом: змінюючи ширину (тривалість) прямокутних імпульсів сигналу визначеної частоти. Сформувати ШІМ-сигнал ми можемо, подавши на певний вивід МК у певній послідовності у часі високі та низькі рівні напруги (одиниці та нулі). Пропустивши отриманий нами ШІМ-сигнал через фільтр низьких частот, на виході фільтра отримаємо рівень напруги, що лінійно пропорційний шпаруватості (відношення тривалості періоду до тривалості імпульсу) імпульсів ШІМ. Призначення фільтра – не пропускати несучу частоту ШІМ. Фільтр може складатися з простої інтегруючої RC-ланки, або бути відсутнім взагалі, якщо кінцеве навантаження має достатню інерцію. Таким чином, маючи у розпорядженні лише два логічні рівня «1» та «0», можна отримати будь-яке проміжне значення аналогового сигналу. За допомогою ШІМ можемо керувати яскравістю освітлення (світлодіоди, лампи розжарення), швидкістю обертання двигунів та формувати звукові сигнали.

У МК AVR формування ШІМ-сигналів є однією з функцій таймерів/лічильників.

8-розрядні МК AVR можуть апаратно генерувати ШІМ-сигнали у двох режимах:

- Fast PWM – швидкодіючий ШІМ;
- Phase Correct PWM – ШІМ з точною фазою;

Режим **Fast PWM** генерує високочастотний ШІМ-сигнал у такий спосіб. Лічильник рахує від нуля до максимального значення, тобто 256, після чого лічильний регістр скидається і цикл повторюється. Якщо відповідний вивід OC_n програмно підключений до таймера, тоді на початку лічби вивід встановлюється в «1» (вис. рівень), а при співпадінні лічильного регістра $TCNT_n$ та регістра порівняння OCR_n вивід таймера/лічильника скидається в нуль (рис. 3.23 а). Також

можемо програмно встановити інверсний режим ШІМ-сигналу, тоді на початку лічби виставляється низький рівень, а при співпадінні лічильного регістра та регістра порівняння встановлюється високий рівень на виході OC_n.

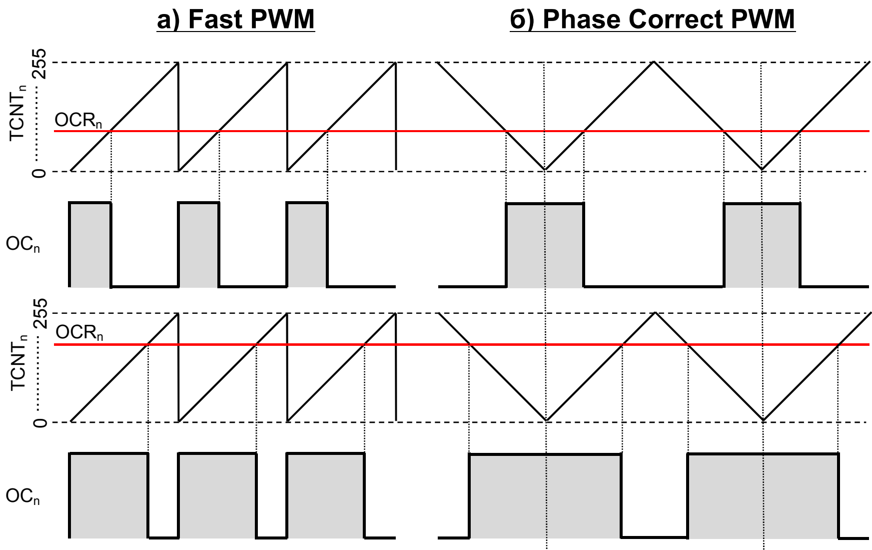


Рис. 3.23. Формування ШІМ-сигналів у режимах Fast та Phase Correct

Для уникнення виникнення несиметричних імпульсів сигналу на виході OC_n передбачена подвійна буферизація запису у регістр OCR_n. Необхідне число спершу зберігається у спеціальному буферному регістрі, а вже запис у регістр порівняння OCR_n здійснюється при досягненні ним значення 255.

Варто звернути увагу на випадок, коли у OCR_n встановлені значення 0 та 255:

- OCR_n=0 – на виводі OC_n при кожному 255+1 такті сигналу таймера буде спостерігатися короткий викид імпульсу. Тому мабуть краще у такому випадку відключати вивід від таймера.
- OCR_n=255 – вивід OC_n буде встановлений у стійкий стан.

Частота ШІМ-сигналу на виводі OC_n розраховується так:

$$f_{OCnPWM} = \frac{f_{\text{такт.генер.Т}n}}{256 \cdot K_{\text{подільн.Т}n}}$$

При 8 МГц тактового генератора МК максимально можемо отримати 31 250 Гц ШІМ-сигнал.

Також для таймера у режимі Fast PWM можуть бути запрограмовані переривання по переповненню та співпадінню. Прапорець TOV_n (по переповненню) виставиться при досягненні лічильником максимального значення (255). Прапорець OCF_n (по співпадінню) виставиться при рівності лічильного регістра та регістра порівняння.

Режим **Phase Correct PWM** генерує ШІМ-сигнали з фіксованою фазою. У цьому режимі лічильник спершу рахує від 0 до 255, а потім здійснює відлік у зворотному порядку від 255 до 0. Таким чином, максимальна частота вихідного сигналу буде у 2 рази меншою, аніж у режимі Fast PWM. При співпадінні лічильного регістра та регістра порівняння у прямому напрямку лічби (від 0 до 255) вихід OC_n скидається у низький рівень, а при співпадінні цих регістрів у зворотному напрямку лічби (від 255 до 0) вивід OC_n встановлюється у високий рівень (рис. 3.23 б). При використанні інверсного ШІМ-сигналу логіка зворотна: спершу встановлюється високий рівень, а потім скидається у низький.

Для запису значення у регістр порівняння OCR_n також використовується попередня буферизація, а запис у цей регістр відбувається лише у момент досягнення лічильником значення 255.

Якщо у регістр порівняння OCR_n записане значення 0 чи 255, то вивід OC_n переключиться в одне зі стійких станів.

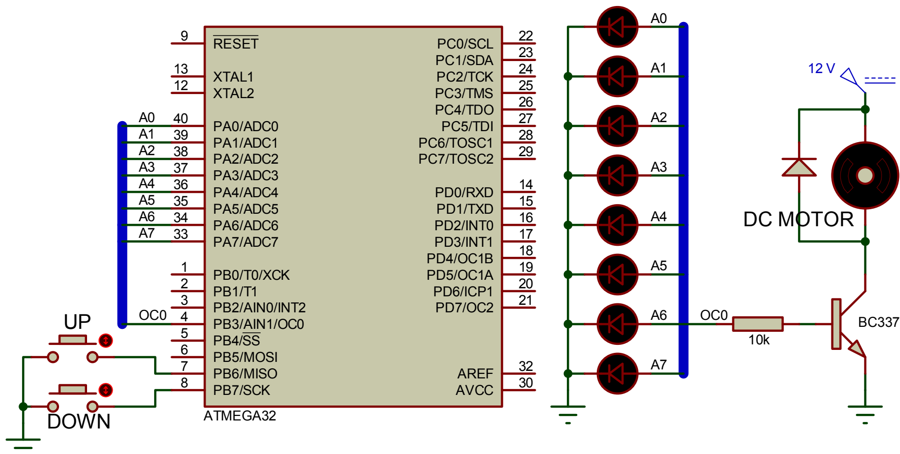
Частота ШІМ-сигналу на виводі OC_n розраховується так:

$$f_{OCnPWM} = \frac{f_{\text{такт.генер.Т}n}}{510 \cdot K_{\text{подільн.Т}n}}$$

Прапорець переривання по співпадінню OCF_n виставляється при кожному співпадінню лічильного регістра та регістра порівняння, а по переповненню TOV_n при досягненні лічильником значення нуль.

Режим Phase Correct PWM використовується для формування багатофазних ШІМ-сигналів (центри яких співпадають у різних каналах), які часто використовуються для керування моторами.

Приклад. Реалізувати керування мотором постійного струму (комп'ютерним вентилятором) за допомогою ШІМ-сигналу у режимі Fast PWM з виводу OC_0 таймера T0. ШІМ-сигнал дискретно розбити на 8 рівнів. Збільшення чи зменшення ширини імпульсу виконувати за допомогою 2-х кнопок, а поточне значення виводити на лінійку з 8-ми світлодіодів, підключених до порту А. Керування мотором виконується за допомогою транзисторного ключа. Для захисту транзисторного ключа паралельно з мотором підключений діод Шоткі.



** на схемі опущені обмежувальні резистори для світлодіодів*

Рис. 3.24. Принципова схема керування мотором постійного струму

```
.include "m32def.inc"
; Імена для регістрів загального призначення--------------------------------------
.def      _temp1      =r16
.def      _temp2      =r17
.def      _temp3      =r18
.def      _power      =r19         ; значення для виводу в OCR0
.def      _leds       =r20         ; значення індикації світлодіодів
.def      _val32      =r21         ; константа, =32

          .CSEG
          ; ініціалізація стека
          ldi              _temp1, Low(RAMEND)
          out              SPL, _temp1
          ldi              _temp1, High(RAMEND)
          out              SPH, _temp1

          ; ініціалізація портів вводу/виводу
          ldi              _temp1, 0x00
          ldi              _temp2, 0xFF
          ; Порт A  працює на вихід
          out              DDRA, _temp2
          out              PORTA, _temp1

          ldi              _temp1, 0x0F
          ldi              _temp2, 0xF0
          ; Порт B – 0..3 вивід на вихід, 4..7 на вхід
          out              DDRB, _temp1
          out              PORTB, _temp2
```

```
        ; ініціалізація таймера T0 -- fast PWM;  fPWM = 31 250 Гц
        ldi     _temp1, (1<<WGM01)|(1<<WGM00)|(1<<CS02)|(1<<COM01)
        out     TCCR0, _temp1          ; подільн=1
        clr     _power
        out     OCR0, _power           ; OCR0 = 0

        clr     _leds                  ; leds = 0
        ldi     _val32, 32             ; val32 = 32

main:
B6:     ; button UP
        sbic    PINB, 6                ; якщо натиснута кнопка UP
        rjmp    B7
        add     _power, _val32         ; power =power + 32
        brcc    B6a                    ; якщо power <= 255
        ldi     _power, 255            ; power = 255
B6a:    out     OCR0, _power           ; OCR0=power
        lsl     _leds                  ; leds = leds <<1
        set                            ; T=1
        bld     _leds,0                ; leds.bit0=1
        out     PORTA, _leds           ; PORTA← leds
        rcall   Pause                  ; виклик підпрограми затримки

B7:     ; button DOWN
        sbic    PINB, 7                ; якщо натиснута кнопка DOWN
        rjmp    end
        sub     _power, _val32         ; power =power - 32
        brcc    B7a                    ; якщо power >= 0
        ldi     _power, 0              ; power = 0
B7a:    out     OCR0, _power           ; OCR0=power
        lsr     _leds                  ; leds = leds >>1
        out     PORTA, _leds           ; PORTA← leds
        rcall   Pause                  ; виклик підпрограми затримки

end:    rjmp    main

; Підпрограма затримки 0,5 сек
Pause:  ldi     _temp1, 0x00
        ldi     _temp2, 0x35
        ldi     _temp3, 0x0C

delay:  subi    _temp1, 1
        sbci    _temp2, 0
        sbci    _temp3, 0
        brne    delay
        ret
```

Оскільки ШІМ-сигнал генерується апаратно, то для логіки кнопок використовуємо просту програмну затримку.

16-розрядні таймери/лічильники МК AVR можуть апаратно генерувати ШІМ-сигнали у трьох режимах:

- Fast PWM – швидкодіючий ШІМ;
- Phase Correct PWM – ШІМ з точною фазою;
- Phase and Frequency Correct PWM – ШІМ з точною фазою та частотою.

На відміну від 8-ми розрядних таймерів/лічильників, 16-розрядні дають певну гнучкість та додаткову функціональність при формуванні ШІМ-сигналів.

Режими Fast PWM і Phase Correct PWM повністю ідентичні аналогічним режимам 8-ми розрядних таймерів/лічильників. Однак у них передбачена можливість генерації ШІМ-сигналів різної розрядності:

- 8 біт, лічба до 255 (0xFF);
- 9 біт, лічба до 511 (0x1FF);
- 10 біт, лічба до 1023 (0x3FF);
- від 2 до 16 біт, лічба від 3 до 0xFFFF (границя лічби фіксується в регістрі OCR_nA або ICR_n).

При роботі з фіксованою частотою ШІМ-сигналу для змінної границі лічби рекомендується використовувати регістр захоплення ICR_n. Однак, якщо часто потрібно змінювати частоту генерованого сигналу, тоді рекомендується використовувати для запису граничного значення лічби регістр порівняння OCR_nA, оскільки регістр захоплення не має подвійної буферизації запису. Це усуває появу несиметричних імпульсів ШІМ-сигналу на виході модулятора. Однак через це регістр OCR_nA не зможе бути використаний для формування ШІМ-сигналу. У цьому випадку ми можемо налаштувати вихід OCnA на генерацію сигналу меандру, записавши у розряди COMnA значення «01».

Режим Phase and Frequency Correct PWM за своєю дією подібний до режиму Phase Correct PWM. Основна його відмінність – це момент оновлення значення регістру порівняння. У цьому режимі оновлення регістру порівняння відбувається у момент досягнення лічильником значення нуля (а не максимального значення). За рахунок цього кожен період сигналу є повністю симетричним.

Звертання до 16-розрядних регістрів. Кожен 16-ти розрядний регістр ($TCNT_n$, OCR_nA, OCR_nB, ICR_n) фізично розміщується у двох 8-ми розрядних регістрах. Відповідно, для доступу до них необхідно здійснювати по дві операції запису чи читання. Для того, щоб запис чи читання обох байтів відбувався одночасно, у складі кожного 16-розрядного таймера є спеціальний 8-ми розрядний регістр TEMP, що призначений для тимчасового зберігання значення старшого байта.

Тому для запису у 16-ти розрядний регістр таймера необхідно спершу виконати запис у старший байт (який поміститься у TEMP), а потім у молодший. При запису значення у молодший, воно об'єднається зі значенням регістра TEMP, та обоє байти одночасно (протягом одного і того ж машинного циклу) запишуться у 16-розрядний регістр.

При читанні послідовність навпаки, спершу зчитується молодший байт. При його читанні значення старшого байта буде записане у регістр TEMP, і при наступному читанні старшого байта буде повернуте значення, що знаходиться у регістрі TEMP.

3.16. Сторожовий таймер (WatchDog).

Сторожовий таймер призначений для захисту мікроконтролера від збоїв та зависань у ході виконання програми. Він має незалежний тактовий генератор, який активний навіть у сплячому режимі мікроконтролера, та працює на частоті в межах 1 МГц.

Якщо сторожовий таймер включений, то через вибрані проміжки часу він виконує скид МК. Тому для нормальної роботи програми цей таймер необхідно регулярно скидати командою wdr через інтервали часу, які менші за його період. Для задання різних періодів спрацювання сторожовий таймер має попередній подільник для власного тактового генератора. Вибір коефіцієнту поділу визначаються бітами WDP у регістрі керування WDTCR.

Таблиця 3.15. Задання періоду сторожового таймера

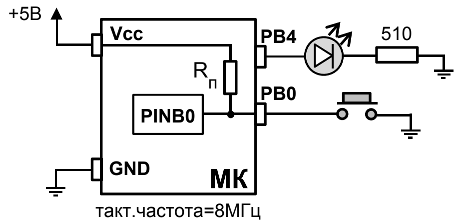
WDP2	WDP1	WDP0	Кількість тактів генератора	Період спрацювання сторож. таймера*
0	0	0	16K (16 384)	16,3 - 17,1 мсек
0	0	1	32K (32 768)	32,5 - 34,3 мсек
0	1	0	64K (65 536)	65,0 - 68,5 мсек
0	1	1	128K (131 072)	0,13 - 0,14 сек
1	0	0	256K (262 144)	0,26 - 0,27 сек
1	0	1	512K (524 288)	0,52 - 0,55 сек
1	1	0	1024K (1 048 576)	1,0 - 1,1 сек
1	1	1	2048K (2 097 152)	2,1 - 2,2 сек

** тривалість спрацювання залежить від напруги живлення, темпер. і т.п.*

Активація сторожового таймера здійснюється бітом WDE у регістрі керування WDTCR («1» – включений).

Вимкнення сторожового таймера виконується послідовністю команд: спершу записати «1» у біти WDE та WDTOE, а потім протягом наступних 4-х машинних циклів записати «0» у біт WDE.

Приклад. Для демонстрації роботи сторожового таймера запрограмуємо кнопку на виконання простої програмної затримки 0,5 сек, а таймер на період спрацювання 0,26-0,27 сек. Одразу після опитування кнопки (та затримки, якщо кнопка натиснута) відбувається скид сторожового таймера та засвічується світлодіод.




```
.include "m32def.inc"
        .CSEG
        ; ініціалізація стека
        ldi     r16, Low(RAMEND)
        out     SPL, r16
        ldi     r16, High(RAMEND)
        out     SPH, r16
        ; Порт B
        ldi     r16, 0xF0
        ldi     r17, 0x0F
        out     DDRB, r16
        out     PORTB, r17
        ; ініціалізація WatchDog (0.26-0.27 sec)
        wdr     ; скид сторожового таймера
        ldi     r16, (1<<WDE)|(1<<WDP2)
        out     WDTCR, r16

main:   sbis    PINB, 0     ; пропустити, якщо кнопка відпущена
        rcall   Pause       ; виклик програми затримки 0,5 сек
        wdr     ; скид сторожового таймера
        sbi     PORTB, 4    ; засвічування світлодіода
        rjmp    main
; Підпрограма затримки 0,5 сек
Pause:  ldi     r16, 0x00
        ldi     r17, 0x35
        ldi     r18, 0x0C
delay:  subi    r16, 1
        sbci    r17, 0
        sbci    r18, 0
        brne    delay
        ret
```

Якщо кнопка натиснута, тоді світлодіод ніколи не засвітиться, бо сторожовий таймер буде здійснювати скид МК ще до завершення підпрограми замтримки 0,5 сек, оскільки він запрограмований на період спрацювання 0,26 сек.

IV. ПРОГРАМУВАННЯ AVR МОВОЮ С

4.1. Основні компілятори Сі для МК AVR.

Існує декілька Сі та С++ компіляторів для МК AVR:

- **AVR Toolchain (WinAVR)** – безкоштовний Сі компілятор для AVR Studio на основі GCC-компілятора. Створює дуже швидкий і компактний код.
- **CodeVisionAVR** – інтегроване середовище розробки програмного забезпечення для мікроконтролерів сімейства Atmel AVR. Представляє собою компілятор мови С, графічну оболонку і генератор шаблонів програм. Крім стандартних бібліотек мови С, компілятор має бібліотеки для роботи з рідкокристалічними індикаторами, різними шинами, деякими датчиками температури, багатьма модулями пам'яті. Також в CodeVisionAVR є автоматичний генератор шаблонів програм, який дає можливість за короткий час отримати готовий код для роботи багатьох функцій МК. Має вбудований програмний модуль для прошивання і конфігурування МК. Комерційний продукт (150 €).
- **IAR Embedded Workbench for AVR** – інтегроване середовище розроблення та налагодження програм для мікроконтролерів AVR з допомогою мови С, С++ і асемблера. У нього входять компілятор мови С і С++, асемблера, компонувальник і відладчик, при цьому можлива взаємодія із зовнішніми програмами типу AVR Studio. Вимагає складного налаштування, не має прикладів в інсталяції й не має генератора початкового коду. Компілятор IAR генерує швидкий і компактний код. Комерційний продукт (1295-1995 €).
- **ImageCraft C for AVR** – інтегроване середовище розробки з оптимізуючим компілятором, розробленим спеціально для AVR. Для ініціалізації периферії містить модуль-генератор Application Builder. Комерційний продукт (250-550 $).
- **MikroC PRO for AVR** – інтегроване середовище розробки з Сі-компілятором стандарту ANSI, широкий набір бібліотек для апаратних засобів. Комерційний продукт (200-250 $).

Курс лекцій у розділі мови Сі зорієнтований суто на синтаксис та бібліотеки безкоштовного компілятора WINAVR, який зараз перейшов «під крило» Atmel та має назву AVR Toolchain.

4.2. Типи даних мови Сі компілятора WINAVR.

У табл. 4.1 представлені числові типи даних, що використовуються у Сі-програмах для компілятора WINAVR. Представлення цілочисельних чисел аналогічне їхньому представленню на асемблері: 0b10101010, 0252, 170, 0xAA.

Таблиця 4.1. Типи даних мови Сі для компілятора WINAVR

Стандартний	Користувацький	К-сть байт	Діапазон значень
char*		1	
signed char	int8_t	1	-128 … 127
unsigned char	uint8_t	1	0 … 255
short		2	-32 768…32 767
unsigned short		2	0 … 65 535
int	int16_t	2	-32 768…32 767
unsigned int	uint16_t	2	0 … 65 535
long	int32_t	4	-2 147 483 648…2 247 483 647
unsigned long	uint32_t	4	0 … 4 294 967 295
long long	int64_t	8	-9 223 372 036 854 775 808 … 9 223 372 036 854 775 807
unsigned long long	uint64_t	8	0..18 446 744 073 709 551 615
float		4	$\pm 1{,}175 \cdot 10^{-38}$ … $\pm 3{,}402 \cdot 10^{38}$
double		8	$\pm 2{,}2 \cdot 10^{-308}$ … $\pm 1{,}8 \cdot 10^{308}$

** знаковий чи беззнаковий цей тип даних визначається опцією компілятора.*

4.3. Бітова арифметика.

а. Бітові поля. Програмуючи мовою асемблер, ми часто використовували окремі байти як резервуари для 8-ми прапорців (бітів), за допомогою яких ми реалізовували логіку програми. Такий підхід давав нам можливість економити регістри загального призначення. Мовою Сі ми також можемо реалізувати такий підхід, працюючи з окремими бітами вибраного байта. Для цього ми оголошуємо певну змінну типу unsigned char та за допомогою порозрядних операцій працюємо з її окремими бітами.

Інший підхід полягає в оголошені структури з бітовими полями.

```
struct Flags
{
        unsigned  f0:1;
        unsigned  f1:1;
        unsigned  f2:1;
        unsigned  f34:2;
        unsigned  f57:3;
};
```

Доступ до окремих бітів байта виконується таким чином:

```
struct Flags cond;
cond.f0=1;
cond.f1=0;
cond.f2=0;
cond.f34=3;          // 0b11
cond.f57=5;          // 0b101
```

Змінна cond розміщується компілятором в оперативній пам'яті SRAM. Тому зміна значень окремих бітів виконується за допомогою порозрядних операцій та числових масок, які обчислюються компілятором.

Згенерований дисасемблером код програми

```
17:       cond.f0=1;
+0000003B:   8189     LDD    R24,Y+1     Load indirect with displacement
+0000003C:   6081     ORI    R24,0x01    Logical OR with immediate
+0000003D:   8389     STD    Y+1,R24     Store indirect with displacement

18:       cond.f1=0;
+0000003E:   8189     LDD    R24,Y+1     Load indirect with displacement
+0000003F:   7F8D     ANDI   R24,0xFD    Logical AND with immediate
+00000040:   8389     STD    Y+1,R24     Store indirect with displacement
```

адреса у пам'яті програм	*код команди*	*назва команди*	*аргументи*	*коментар*

б. Порозрядні операції. У алгоритмічній мові Сі для роботи з цілими числами у двійковому представленні є шість порозрядних операцій: & (І), | (АБО), ^ (виключне АБО чи сума за модулем 2), ~ (НЕ), >> (зсув вправо) та << (зсув вліво). Результат їх виконання для двох бітів наведений у табл. 4.2 та табл. 4.3.

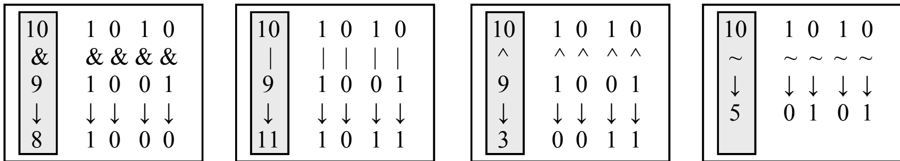
Таблиця 4.2. Таблиця істинності для порозрядних операцій

		Результат			**~**	
Біт 1	**Біт 2**	**&**	**|**	**^**	**Біт**	**Результат**
0	0	0	0	0	0	1
0	1	0	1	1	1	0
1	0	0	1	1		
1	1	1	1	0		

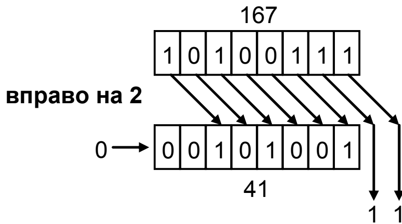
Таблиця 4.3. Порозрядні операції зсуву

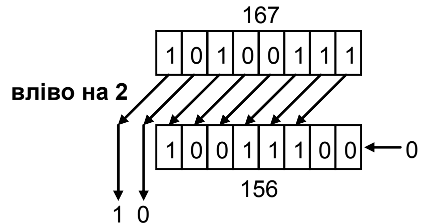
>> зсув вправо	Зсуває біти лівого операнда на число розрядів, що вказане правим операндом. При цьому праві біти втрачаються. Якщо лівий операнд представляє собою ціле число без знаку, то ліві біти, що звільнилися, заповнюються нулями, в іншому випадку, вони заповнюються символом знаку, тобто 1.
<< зсув вліво	Зсуває біти лівого операнда на число розрядів, що вказане правим операндом. При цьому ліві біти втрачаються, а праві заповнюються нулями.

Результати цих операцій для цілих чисел виглядають так:

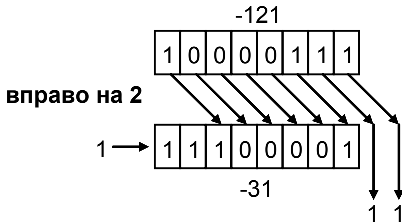


Порозрядний зсув беззнакового однобайтного числа 167





Порозрядний зсув знакового однобайтного числа -121



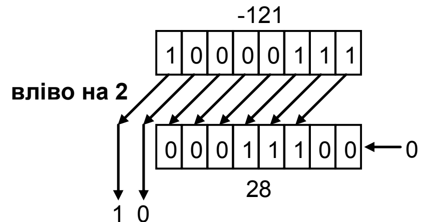


Рис. 4.1. Ілюстрація роботи порозрядних зсувів

Операція порозрядного зсуву вліво << цілого числа на *n* розрядів еквівалентна множенню числа на 2^n, при умові що крайні біти не губляться. Наприклад, при зсуві числа 5 (101_2) на 3 розряди вліво отримуємо число 40 (101000_2), що тотожно множенню числа 5 на 8 = 2^3.

Операція порозрядного зсуву вправо >> цілого числа на n розрядів еквівалентна цілочисельному діленню цього числа на 2^n. Наприклад, при зсуві числа 53 (110101_2) на 3 розряди вправо отримуємо число 6 (110_2), що тотожно цілочисельному діленню числа 53 на 8 = 2^3 (53 / 8 = 6,625).

Зазначимо, що біти нумеруються справа наліво, причому крайній справа (молодший) біт має номер 0. Довжина біта, півбайта, байта, півслова, слова та подвійного слова рівні, відповідно, 1, 4, 8, 16, 32 та 64 (рис. 4.2).

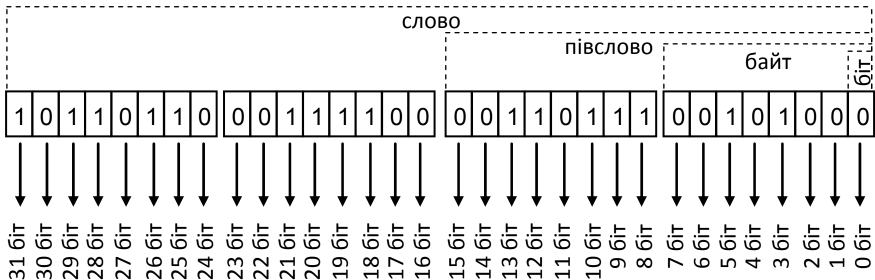


Рис. 4.2. Внутрішня структура цілого числа та нумерація його бітів

в. Встановлення чи очищення бітів. При роботі з регістрами вводу/виводу (периферії) часто необхідно встановлювати чи скидати окремі біти, чи навіть декілька бітів одночасно. На асемблері для роботи з окремими бітами були відповідні команди sbi та cbi. Рекомендується використовувати для встановлення та скидання окремих бітів такі конструкції мови Сі.

```
unsigned char  C=0b10101010;              // глобальна змінна
```

Встановлення окремого біта в «1»:

```
C |= (1<<6);          // встановлення 6-го біта в "1";   C=0b11101010

LDS     R24,0x0060    Load direct from data space
ORI     R24,0x40      Logical OR with immediate
STS     0x0060,R24    Store direct to data space
```

Згенерований дисасемблером код

Скидання окремого біта в «0»:

```
C &= ~(1<<3);         // скидання 3-го біта в "0";   C=0b11100010

LDS     R24,0x0060    Load direct from data space
ORI     R24,0x40      Logical OR with immediate
STS     0x0060,R24    Store direct to data space
```

Згенерований дисасемблером код

Аналогічні маніпуляції і для групи бітів:

```
C |= (1<<4)|(1<<2)|(1<<0);           // встановлення 0,2,4 біта в "1"
C &= ~( (1<<4)|(1<<2)|(1<<0) );      // скидання 0, 2, 4 біта в "0"
```

При роботі з портами вводу/виводу компілятор генерує для встановлення/скидання окремих бітів більш компактний код:

```
DDRA |= (1<<6);                      // встановлення 6-го біта в "1";

SBI     0x1A,6      Set bit in I/O register

DDRA &= ~(1<<3);                     // скидання 3-го біта в "0";

CBI     0x1A,3      Clear bit in I/O register

DDRA |= (1<<4)|(1<<2)|(1<<0);        // встановлення 0, 2, 4 біта в "1"

IN      R24,0x1A    In from I/O location
ORI     R24,0x15    Logical OR with immediate
OUT     0x1A,R24    Out to I/O location

DDRA &= ~((1<<4)|(1<<2)|(1<<0)); // скидання 0, 2, 4 біта в "0"

IN      R24,0x1A    In from I/O location
ANDI    R24,0xEA    Logical AND with immediate
OUT     0x1A,R24    Out to I/O location
```

г. Інверсія бітів. Для інверсії бітів використовується операція виключного АБО (сума за модулем 2).

```
C ^= (1<<2);                         // інверсія 2-го біта змінної C

LDS     R24,0x0060    Load direct from data space
LDI     R18,0x04      Load immediate
EOR     R24,R18       Exclusive OR
STS     0x0060,R24    Store direct to data space

C ^= (1<<4)|(1<<2)|(1<<0);           // інверсія 0,2,4 біта змінної C
```

Аналогічно і для регістрів вводу/виводу:

```
DDRA ^= (1<<2);                      // інверсія 2-го біта регістра DDRA

IN      R24,0x1A    In from I/O location
LDI     R25,0x04    Load immediate
EOR     R24,R25     Exclusive OR
OUT     0x1A,R24    Out to I/O location

DDRA ^= (1<<4)|(1<<2)|(1<<0);        // інверсія 0,2,4 біта регістра DDRA
```

д. Перевірка значень бітів. В умовних конструкціях часто необхідно виконувати перевірку значень окремих бітів цілочисельного числа чи регістра вводу/виводу. Для цього ми можемо скористатися або готовим макросом WINAVR, або написати відповідну конструкцію мовою Сі.

```
unsigned char  ivalue = 121;       // глобальна змінна


// якщо 2-й біт у змінній ivalue встановлений, bit2<>0
if( bit_is_set(ivalue, 2) )  ivalue=10;          // використовуючи макрос bit_is_set()

if( ivalue & (1<<2) )        ivalue=10;          // використовуючи Сі-конструкцію

LDS     R24,0x0061     Load direct from data space
SBRS    R24,2          Skip if bit in register set
RJMP    PC+0x0004      Relative jump
LDI     R24,0x0A       Load immediate
STS     0x0061,R24     Store direct to data space

// якщо 2-й біт у змінній ivalue скинутий, bit2=0
if( bit_is_clear(ivalue, 2) )  ivalue=10;        // використовуючи макрос bit_is_clear()

if( !(ivalue & (1<<2)) )       ivalue=10;        // використовуючи Сі-конструкцію

LDS     R24,0x0061     Load direct from data space
SBRC    R24,2          Skip if bit in register cleared
RJMP    PC+0x0004      Relative jump
LDI     R24,0x0A       Load immediate
STS     0x0061,R24     Store direct to data space


// якщо 2-й біт у регістрі DDRA встановлений, bit2<>0
if( bit_is_set(DDRA, 2) ) DDRA=10;               // використовуючи макрос bit_is_set()

if( DDRA & (1<<2) )       DDRA=10;               // використовуючи Сі-конструкцію

SBIS    0x1A,2         Skip if bit in I/O register set
RJMP    PC+0x0003      Relative jump
LDI     R24,0x0A       Load immediate
OUT     0x1A,R24       Out to I/O location

// якщо 2-й біт у регістрі DDRA скинутий, bit2=0
if( bit_is_clear(DDRA, 2) ) DDRA=10;             // використовуючи макрос bit_is_clear()

if( !(DDRA & (1<<2)) )      DDRA=10;             // використовуючи Сі-конструкцію

SBIC    0x1A,2         Skip if bit in I/O register cleared
RJMP    PC+0x0003      Relative jump
LDI     R24,0x0A       Load immediate
OUT     0x1A,R24       Out to I/O location
```

Умова в операторі **if**() справджується, якщо її результат відмінний від нуля, і навпаки, якщо вираз умови повернув значення нуль, тоді умова не справджується. Булеві типи на мові Сі відсутні. Таким чином, ми можемо формувати логічні вирази для перевірки значень одразу декількох бітів. Наприклад:

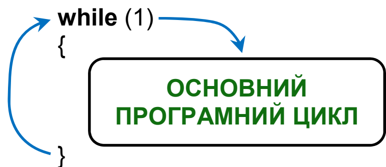
```
// якщо 1й або 3й біти у змінній ivalue встановлені, (bit1==1 || bit3==1)
if( ivalue & 0b00001010 )  ivalue=10;

// якщо 1й та 3й біти у змінній ivalue встановлені, (bit1==1 && bit3==1)
if( ivalue & 0b000001010 == 0b00001010)  ivalue=10;


// якщо 1й або 3й біти у змінній ivalue скинуті, (bit1==0 || bit3==0)
if( ~ivalue & 0b00001010)  ivalue=10;

// якщо 1й та 3й біти у змінній ivalue скинуті, (bit1==0 && bit3==0)
if( !(ivalue & 0b000001010) )  ivalue=10;
```

Маска 0b00001010 відповідає виразу (1<<3)|(1<<1).

4.4. Базова структура програми мовою Сі.

```
// перелік підключених бібліотек
#include <avr/io.h>
#include <avr/interrupt.h>


// секція для макровизначень
#define XTAL 8000000L
#define HI(x) ((x) >> 8)              // макрос для отримання ст. байта
#define LO(x) ((x) & 0xFF)            // макрос для отримання мол. байта


// оголошення структур даних та глобальних змінних
struct Time
{
        unsigned char  hour;
        unsigned char  minute;
        unsigned char  second;
};

volatile unsigned char  temp;
int  value;


// оголошення прототипів користувацьких функцій
void  Init(void);
```

```
// підпрограми переривань
ISR(TIMER0_OVF_vect)            // переривання Таймера Т0 по переповненню
{

}


// основна функція
int  main(void)
{
        // оголошення та ініціалізація локальних змінних
        struct Time time1={0, 0, 0};
        int  k=0;
        // ініціалізація периферії
        Init();

        while (1)
        {
```




```
        }
}


// користувацькі функції
void  Init(void)
{
        DDRA  = 0x00;           // Порт А на вхід
        PORTA = 0xFF;           // з внутр. підтягуючими резисторами

        DDRB  = 0xFF;           // Порт В на вихід
        PORTB = 0x00;           // низький рівень
}
```

У налаштуваннях проекту AVR Studio v.4 на основі компілятора мови Сі (WINAVR) обов'язково необхідно вказувати назву МК та частоту тактового генератора, на якій він буде працювати.

У кожному проекті обов'язково слід підключати бібліотеку <avr/io.h>. Завдяки їй компілятор виконає для нашого конкретного МК імпорт усіх назв регістрів вводу/виводу з їхніми адресами, а також назви окремих бітів, що розміщуються у цих регістрах.

Якщо у програмі використовуються переривання, тоді також слід підключити бібліотеку <avr/interrupt.h>.

Макрос ISR() вказує на підпрограму переривання. Як аргумент макросу вказується назва переривання.

Основна функція main (з якої починається робота МК), містить необхідну ініціалізацію периферії та основний робочий безмежний цикл. Як правило, його реалізують у такому вигляді:

```
while (1)
{

}
```

Альтернативним варіантом може бути і такий безмежний цикл

```
for ( ; ; )
{

}
```

Користувацькі функції можемо вказувати як перед основною функцією main, так і після неї. У другому випадку, щоб не було повідомлень компілятора, необхідно перед функцією main оголошувати прототипи наших користувацьких функцій.

Примітка: ініціалізацію стеку компілятор виконує автоматично.

4.5. Глобальні та локальні змінні, директива volatile.

Змінні, що оголошені у глобальному просторі програми, розміщуються компілятором в оперативній пам'яті SRAM, починаючи з адреси $060. Локальні змінні, оголошені усередині функцій та підпрограм переривань, розміщуються компілятором у стеку.

Підпрограми переривань не приймають ніяких аргументів, тому для передачі їм значень із основної функції та від них в основну функцію, слід використовувати глобальні змінні.

Важливою рисою компілятора Сі є те, що він виконує оптимізацію коду, написаного програмістом, щоб кінцевий згенерований код на асемблері був коротким і швидкодіючим. Однак оптимізація, що виконується компілятором, може спотворити логіку програми, а навіть і зробити її взагалі непрацездатною.

При оптимізації та спрощенню коду компілятор часто викидає куски програмного коду, які на його думку не впливають на роботу МК. Тобто, компілятор може вилучити всі ділянки програми, які опрацьовують змінні, що не змінюють (на думку компілятора) своїх значень.

Наприклад, ми хочемо зробити невеличку програмну затримку:

```
int  main(void)
{
          for(int i=0; i<10000; i++) ;
          while (1) {  }
}
```

Компілятор розгляне цикл for як такий, що не впливає на роботу МК, і тому він його вилучить на етапі компіляції.

Інший випадок, ми оголошуємо певну глобальну змінну, яка буде використовуватися як в основній програмі, так і в перериванні. В основній програмі ми присвоюємо їй значення нуль, і одразу після цього перевіряємо її у циклі while на рівність нулю. Допоки змінна рівна нулю – цикл має постійно повторюватися, аж до моменту зміни її значення на одиницю, при настанні події переривання таймера T0 по переповненню.

```
unsigned char  temp;

ISR(TIMER0_OVF_vect)   // переривання Таймера Т0 по переповненню
{
          temp = 1;
}
int main(void)
{         // ініціалізація таймера T0
          TCCR0 = (1<<CS02)|(1<<CS00);   // Кпод=1024, по переповненню
          TIMSK |= 1<<TOIE0;        // дозвіл на переривання по переповненню
          sei();                    // глобальний дозвіл на переривання

          temp = 0;
          while (temp == 0);        // тут програма зациклиться на постійно

          DDRA = 0xF0;              // компілятор пропустить цей код
          PORTA= 0x0F;              // компілятор пропустить цей код
          while (1)  {  }           // компілятор пропустить цей код
}
```

Оскільки змінна temp ініціалізується нулем безпосередньо перед циклом while, а цикл повторюється до тих пір, поки ця змінна рівна нулю, а всередині циклу змінна не змінюється, то компілятор робить відповідний висновок, що тут передбачене постійне зациклення. Все що нижче циклу (ініціалізація порту А і далі) компілятором ігнорується. Хоча в дійсності ми передбачили, що змінна temp змінить своє значення у перериванні таймера, і відбудеться вихід з циклу.

Це пояснюється тим, що компілятор не аналізує код у паралельних процесах (основна програма та підпрограми переривань). Тому такі змінні слід оголошувати з ключовим словом **volatile** (з англ. означає «непостійний»).

Директива volatile у мові Сі вказує компілятору, що значення зазначеної змінної може бути зміненим у будь-який момент, навіть нехай і невідомим для компілятора способом, і що частина коду, яка виконує над цією змінною певні дії (читання чи запис), не повинна бути оптимізованою. Тобто, оголошену таким чином змінну, не можна чіпати при оптимізації.

Для коректної роботи попередніх прикладів необхідно відповідні змінні оголосити з ключовим словом volatile.

```
//======= Для першого прикладу ==========
for(volatile int i=0; i<10000; i++) ;

//======= Для другого прикладу ===========
volatile unsigned char  temp;
```

4.6. Робота з перериваннями.

Підпрограма переривання визначається за допомогою макросу ISR(). Цей макрос, згідно вказаної у дужках назви вектора переривань, реєструє та позначає вказану підпрограму як обробник переривання для визначеного периферійного пристрою. Наступний приклад ілюструє визначення обробника події для переривання від модуля АЦП.

```
#include <avr/interrupt.h>

ISR(ADC_vect)
{
   // тут вписується користувацький код
}
```

При використанні переривань обов’язково необхідно підключати бібліотеку <avr/interrupt.h>. Ця бібліотека визначає для вказаної моделі МК необхідні адреси векторів переривань.

Якщо переривання для певного периферійного пристрою дозволене, але немає визначеного обробника для цього переривання (слід розглядати це як програмний дефект), тоді при виникненні цієї події буде виконаний скид МК та перенаправлення на вектор RESET.

Для підпрограми обробки події переривання компілятор автоматично генерує код для кешування у стеку регістра стану SREG на

початку підпрограми та його відновлення зі стеку при виході з підпрограми переривання.

У табл. 4.4 вказаний перелік усіх назв векторів переривань для моделі ATmega32. Ці назви вписуються як аргумент макросу ISR().

Таблиця 4.4. Перелік назв векторів переривань для ATmega32

Назва вектора	Опис
ADC_vect	ADC Conversion Complete
ANA_COMP_vect	Analog Comparator
EE_RDY_vect	EEPROM Ready
INT0_vect	External Interrupt Request 0
INT1_vect	External Interrupt Request 1
INT2_vect	External Interrupt Request 2
SPI_STC_vect	Serial Transfer Complete
SPM_RDY_vect	Store Program Memory Ready
TIMER0_COMP_vect	Timer/Counter0 Compare Match
TIMER0_OVF_vect	Timer/Counter0 Overflow
TIMER1_CAPT_vect	Timer/Counter1 Capture Event
TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
TIMER1_COMPB_vect	Timer/Counter1 Compare MatchB
TIMER1_OVF_vect	Timer/Counter1 Overflow
TIMER2_COMP_vect	Timer/Counter2 Compare Match
TIMER2_OVF_vect	Timer/Counter2 Overflow
TWI_vect	2-wire Serial Interface
USART_RXC_vect	USART, Rx Complete
USART_TXC_vect	USART, Tx Complete
USART_UDRE_vect	USART Data Register Empty

Для надання дозволу глобального переривання чи його заборони використовуються відповідні макроси sei() та cli(). Також у бібліотеці avr/interrupt.h вказуються й інші макроси для організації переривань.

4.7. Робота з даними в пам'яті програм.

У бібліотеці <avr/pgmspace.h> визначаються функції та макроси, що дають можливість звертатися до даних, які зберігаються у пам'яті програм, тобто сегменті коду.

Для того, щоб вказати, що певна змінна, масив даних чи об'єкт визначеної структури розміщується в області пам'яті програм, необхідно в описі типу даних використати макрос PROGMEM та проініціалізувати значеннями. При оголошенні даних без цього атрибута, ці дані будуть розміщені в оперативній пам'яті.

```
PROGMEM int A[2][2] = {{1,2},{3,4}};
PROGMEM char Lecturer[] = "Pavelchak Andriy";
PROGMEM int num = 0b10101010;
PROGMEM float fvalue = 22.2;
struct Struc
{
        int a;
        char b;
};
PROGMEM struct Struc str_mas[2] = {{22, 55}, {33, 44}};
```

Атрибут PROGMEM може розміщуватися у довільному місці при оголошенні змінної:

```
int PROGMEM A[2][2]={{1,2},{3,4}};
int A[2][2] PROGMEM ={{1,2},{3,4}};
```

Доступ до значень об'єктів, розміщених в області пам'яті програм, виконується лише за допомогою таких макросів

```
pgm_read_byte(адреса)
pgm_read_word(адреса)
pgm_read_dword(адреса)
pgm_read_float(адреса)
```

Наприклад, при звертанні до елементів структури

```
PORTB = pgm_read_word(&str_mas[0].a);
PORTD = pgm_read_byte(&str_mas[0].b);
```

Приклад. Реалізуємо на основі лінійки з 8-ми світлодіодів, підключених до порту C, різні алгоритми їхнього засвічування: 1) 0→1→2→3→4→5→6→7; 2) 7→6→5→4→3→2→1→0; 3) 7&0→6&1→5&2 →4&3; 4) 4&3→5&2→6&1→7&0; 5) усі разом блимають. Зміна алгоритму виконується кнопкою, підключеною до виводу PB0.

```
#include <avr/interrupt.h>
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

const unsigned char Alg[5][8] PROGMEM = {
{1<<0, 1<<1, 1<<2, 1<<3, 1<<4, 1<<5, 1<<6, 1<<7},               // алг. 1
{1<<7, 1<<6, 1<<5, 1<<4, 1<<3, 1<<2, 1<<1, 1<<0},               // алг. 2
{1<<7|1<<0, 1<<6|1<<1, 1<<5|1<<2, 1<<4|1<<3,
          1<<7|1<<0, 1<<6|1<<1, 1<<5|1<<2, 1<<4|1<<3},          // алг. 3
```

```
{1<<3|1<<4,  1<<2|1<<5,  1<<1|1<<6,  1<<0|1<<7,
                    1<<3|1<<4,  1<<2|1<<5,  1<<1|1<<6,  1<<0|1<<7 },   // алг. 4
{0xFF,  0,  0xFF,  0,  0xFF,  0,  0xFF,  0} };                          // алг. 5

int main(void)
{
        DDRB = 0x00;  PORTB = 0xFF;      // ініціалізація порту B
        DDRC = 0xFF; PORTC = 0x00;       // ініціалізація порту C
        unsigned char  N_alg = 0;        // визначає № алгоритму
        while (1)
        {
                for(int i=0; i<8; i++)
                {       // якщо кнопка натиснута, то зміна алгоритму
                        if( !(PINB & (1<<0)) )  {N_alg++;  _delay_ms(500);}
                        if(N_alg==5)  N_alg = 0;
                        // вивід у порт значення з пам'яті програм
                        PORTC = pgm_read_byte( &Alg[N_alg][i] );
                        _delay_ms(500);
                }
        }
}
```

4.8. Програмні затримки.

Для організації затримок за допомогою програмних циклів використовуються функції бібліотеки <util/delay.h>. Згідно вхідного параметра функції на етапі компіляції обчислюється необхідна кількість пустих програмних циклів, які, згідно заданої тактової частоти, слід виконати, щоб отримати вказану затримку. Тому перед використанням модуля<util/delay.h> необхідно забезпечити такі 2 умови:

– Визначити значення константи F_CPU, що рівне тактовій частоті контролера у герцах.
– Включити оптимізацію при компіляції.

Варто зазначити, що тривалість цих затримок може бути більшою, через можливі переходи на виконання переривань.

Функції модуля:

void _delay_ms(**double** ms) – виконує затримку у ms мілісекунд. Максимально можливе значення (262.14 мсек) / (F_CPU у МГц). Якщо вхідний параметр більший за це значення, тоді відбудеться автоматичне зниження точності витримки інтервалів затримки. Таким чином можна реалізувати затримку до 530 сек (для 8 МГц).

void _delay_us(**double** us) – виконує затримку у us мікросекунд. Максимально можливе значення (768 мксек) / (F_CPU у МГц). Якщо значення затримки виходить більшим, тоді виклик функції автоматично буде перенаправлений у функцію _delay_ms().

4.9. Організація передачі даних через UART/USART.

Універсальний синхронний/асинхронний послідовний прийомо-передавач (USART) забезпечує обмін даними МК AVR з зовнішніми пристроями по послідовному каналу в повнодуплексному режимі. При цьому передача даних може бути як асинхронна так і синхронна.

При **синхронному** послідовному вводі/виводі передача окремих бітів даних синхронізується за допомогою тактового сигналу, який передається одночасно з даними. Синхронна послідовна передача даних використовується, основним чином, на рівні друкованих плат, у тому числі, для обміну даними між різними інтегрованими блоками у складі схеми МК та різними периферійними схемами.

При **асинхронній** передачі даних синхронізація виконується у часі за допомогою стартових та стопових бітів, що визначають початок та кінець передачі слова даних. Асинхронна передача даних використовується для комунікації блоків, розділених у просторі та які мають певну автономність один від одного. Наприклад, між ПК та принтером, між пристроєм на базі МК та комп'ютером.

Модуль USART підтримує як синхронний, так і асинхронний режими роботи. Однак на практиці його найчастіше використовують саме в асинхронному режимі, а синхронний режим реалізують за допомогою модуля SPI. Тому в курсі лекцій ми розглядатимемо лише асинхронний режим, і надалі модуль називатимемо UART.

а. Формат передачі кадру даних UART. За своєю структурою він ідентичний інтерфейсу RS-232, з тією лиш відмінністю, що в інтерфейсі RS-232 логічні рівні формуються напругами від ±3 до ±12В, а в модулі UART логічні рівні відповідають TTL-рівням (0 та 5 В).

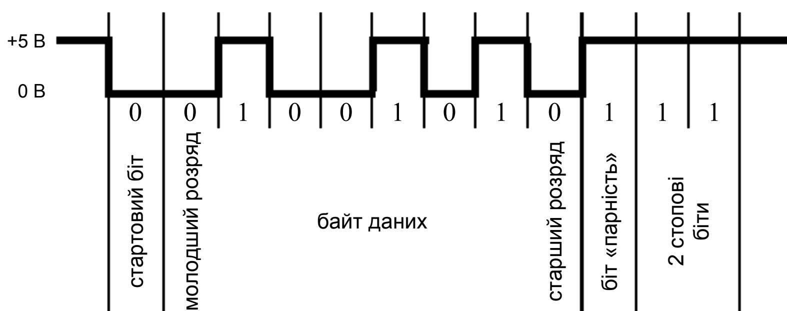


Рис. 4.3. Представлення числа 82 (01010010_2)
у кадрі 8n2 з перевіркою на парність

Початок кадру даних завжди фіксується низьким рівнем стартового біту (рис. 4.3). Після цього йде байт даних (5-9 бітів) з молодшими розрядами спереду. Якщо дозволена перевірка на парність, то далі йде біт парності, що доповнює байт даних «1» чи «0» так, щоб кількість «1» байту даних була парною (при опції «Парність») чи непарною (при опції «Непарність»). Цей простий засіб дає можливість виявляти непарну кількість спотворених бітів. Останніми передаються стопові (1 чи 2) біти, що представлені високим рівнем. Якщо на лінії передача даних відсутня, тоді на ній завжди присутній високий рівень.

Швидкість передачі вимірюється у бодах (baud), бітів за секунду (bps), і на рис. 4.3 кадр даних складається з 12 бодів.

Формат кадру задається відповідними бітами регістрів керування UCSRB та UCSRC.

Таблиця 4.5. Визначення розміру байту даних

	Кількість біт у байті даних				
	5 біт	6 біт	7 біт	8 біт	9 біт
UCSZ0	0	1	0	1	1
UCSZ1	0	0	1	1	1
UCSZ2	0	0	0	0	1

Таблиця 4.6. Керування контролем парності

	немає	парність	непарність
UPM0	0	0	1
UPM1	0	1	1

Вибір кількості стоп-бітів здійснюється за допомогою розряду USBS у регістрі керування UCSRC. Якщо цей розряд скинутий в «0», тоді передавач формує 1 стоп-біт у кінці посилки. Якщо ж встановлений в «1», тоді – 2 стоп-біти. Варто зазначити, що приймачем другий стоп-біт ігнорується, і відповідно, помилки кадрування виявляються лише для першого стоп-біта. Найбільш популярним є формат кадру 8n1 (1 старт, 8 біт даних, 1 стоп) без контролю парності.

б. Підключення UART. У МК AVR протокол передачі даних UART реалізований апаратно. На деяких моделях навіть є реалізовано 2 модулі UART. Приймач даних під'єднаний до виводу з надписом RxD, а передавач до виводу TxD. При з'єднанні між собою двох модулів UART для передачі даних, необхідно з'єднати навхрест між собою виводи модулів передачі та приймання, як на рис. 4.4.

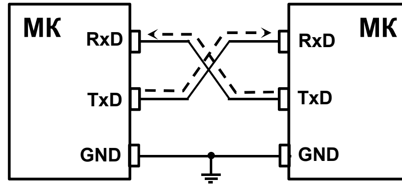


Рис. 4.4. З'єднання модулів UART для передачі даних

в. Апаратна частина UART. Модуль складається з 3-х основних частин (рис. 4.5): тактового генератора (контролера) швидкості передачі, блоку приймача та блоку передавача.

Блок передавача містить однорівневий буфер, зсувний регістр, схему формування біта парності та схему керування. Блок ж приймача містить схеми відновлення тактового сигналу та даних, схему контролю парності, дворівневий буфер, зсувний регістр та схему керування.

Буферні регістри приймача та передавача розміщуються за єдиним адресом простору вводу/виводу та позначаються як регістр даних UDR. У цьому регістрі зберігаються молодші 8 розрядів даних, що приймаються чи передаються. При читанні UDR виконується звертання до буферного регістра приймача, а при записі – до буферного регістра передавача.

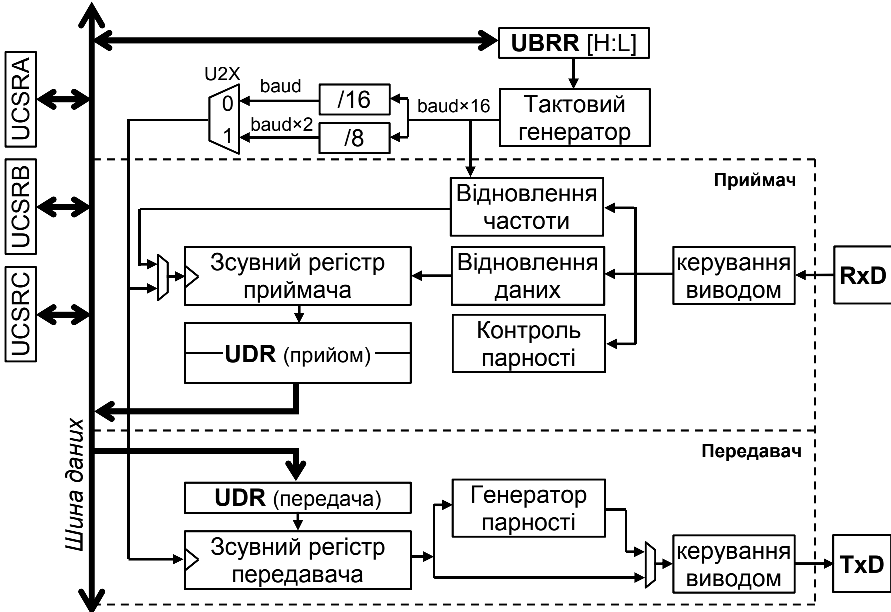


Рис. 4.5. Спрощена структурна схема модуля UART

У модулях USART буфер приймача дворівневий (FIFO-буфер). При будь-якому звертанні до регістра UDR цей буфер змінює свій стан. Тому необхідно спершу зчитати дані з цього регістра, а потім вже виконувати необхідні маніпуляції над ним.

Регістр контролера швидкості UBRR задає необхідний коефіцієнт поділу для системного тактового сигналу, після чого цей сигнал ще поступає на додаткові дільники, вибір яких здійснюється за допомогою додаткового біта U2X.

Схема відновлення тактового сигналу (у приймачі) призначена для синхронізації внутрішнього тактового сигналу, що формується контролером швидкості передачі, та пакетів з даними, що поступають на вивід RxD. Схема відновлення даних виконує зчитування та фільтрацію кожного розряду для отримуваного пакету.

г. Швидкість прийому/передачі. Швидкість обміну задається контролером швидкості передачі, що функціонує як подільник системного тактового сигналу з програмованим коефіцієнтом поділу, значення якого знаходиться у регістрі UBRR. Регістр UBRR є 12-розрядний та фізично розміщується у 2-х регістрах UBRRH та UBRRL.

В асинхронному режимі швидкість обміну визначається не лише значенням регістра UBRR, але і станом розряду U2X у регістрі керування UCSRA. Якщо цей біт встановлений в «1», то коефіцієнт поділу подільника зменшується у 2 рази, а швидкість, відповідно, подвоюється. Швидкість обміну в асинхронному режимі визначається за такими формулами:

$$\text{при U2X} = 0: \quad \mathbf{BAUD} = \frac{\text{XTAL}}{16(\text{UBBR} + 1)}; \ \mathbf{UBRR} = \frac{\text{XTAL}}{16 \cdot \text{BAUD}} - 1$$

$$\text{при U2X} = 1: \quad \mathbf{BAUD} = \frac{\text{XTAL}}{8(\text{UBBR} + 1)}; \ \mathbf{UBRR} = \frac{\text{XTAL}}{8 \cdot \text{BAUD}} - 1$$

Прийняті такі стандартні швидкості обміну даними: 1200, 1800, 2400, 4800, 7200, 9600, 14400, 19200, 28800, 38400, 57600, 76800, 115200, 230400 бод.

Для уникнення виникнення помилок передачі рекомендується використовувати стабілізований кварцовий тактовий генератор. Також має значення і величина частоти, на якій працює кварцовий кристал (табл. 4.5). На деяких частотах можна отримати нульову похибку при передачі даних відносно ряду стандартних швидкостей. Похибка передачі обчислюється за такою формулою:

$$\text{Error}[\%] = \left(\frac{\text{BAUD}_{\text{розрах.}}}{\text{BAUD}} - 1\right) \cdot 100\%.$$

Розрахуємо похибку для стандартної швидкості 9600 бод при частоті тактового генератора 8МГц.

$$\text{UBRR} = \frac{8 \cdot 10^6}{16 \cdot 9600} - 1 = 51{,}083 = 51;$$

$$\text{BAUD}_{\text{розрах.}} = \frac{8 \cdot 10^6}{16(51 + 1)} = 9615{,}38 \text{ Бод};$$

$$\text{Error}[\%] = \left(\frac{9615{,}38}{9600} - 1\right) \cdot 100\% = 0{,}16\%.$$

Рекомендується використовувати значення регістра UBRR, при яких отримана швидкість передачі відрізняється від необхідного значення менше, аніж на 0,5%.

Таблиця 4.7. Оцінка похибки передачі при швидкості 9600 бод

	1 МГц		3.6864 МГц		4 МГц		6 МГц	
	UBRR	**Error**	**UBRR**	**Error**	**UBRR**	**Error**	**UBRR**	**Error**
U2X=0	0x06	-7,0%	0x17	0%	0x19	0,16%	0x26	0,16%
U2X=1	0x0C	0,16%	0x2F	0%	0x33	0,16%	0x4D	0,16%

	7.3728 МГц		8 МГц		9.216 МГц		10 МГц	
	UBRR	**Error**	**UBRR**	**Error**	**UBRR**	**Error**	**UBRR**	**Error**
U2X=0	0x2F	0%	0x33	0,16%	0x3B	0%	0x40	0,16%
U2X=1	0x5F	0%	0x67	0,16%	0x77	0%	0x81	0,16%

	11.0592 МГц		12 МГц		14.7456 МГц		16 МГц	
	UBRR	**Error**	**UBRR**	**Error**	**UBRR**	**Error**	**UBRR**	**Error**
U2X=0	0x47	0%	0x4D	0,16%	0x5F	0%	0x67	0,16%
U2X=1	0x8F	0%	0x9B	0,16%	0xBF	0%	0xCF	0,16%

**похибка визначає відхилення швидкості передачі*

Варто звернути увагу, що на частотах тактового генератора рівних 3.6864, 7.3728, 11.0592 та 14.7456 МГц похибка передачі є рівною нулю для усієї лінійки стандартних швидкостей.

Зауваження. Для деякий моделей AVR адреса старшого байта UBRRH суміщена з адресою регістра керування UCSRC. Тому при записі необхідний регістр визначається станом старшого розряду записуваного значення. Якщо старший біт скинутий в «0», то значення записується у регістр UBRRH, а якщо встановлений в «1», тоді у регістр керування UCSRC.

д. Передача та прийом даних, переривання модуля UART.

Для активації прийому/передачі модуля UART необхідно надати дозволи на роботу передавача та приймача, встановивши відповідні біти TXEN та RXEN у регістрі керування UCSRB. Тоді відповідні виводи МК, позначені як TxD та RxD, підключаються до модуля UART та працюють на прийом і передачу, незалежно від налаштувань регістрів керування портом, до якого вони належать.

Для відправки байту даних необхідно записати його значення у регістр даних UDR. Після цього ці дані пересилаються із UDR у зсувний регістр передавача. Якщо в регістр UDR відправити одразу ще один байт даних, то ці дані будуть відправлені у зсувний регістр лише після того, як у зсувному регістрі буде відправлений останній біт з кадру. Отже, частота запису даних в UDR визначається швидкістю обміну даними модуля UART.

Прийом даних починається з моменту виявлення приймачем коректного старт-біту. Далі, кожен наступний біт кадру зчитується зі швидкістю, заданою для модуля UART, та розміщується у зсувному регістрі, аж поки не буде виявлений перший стоп-біт. Після цього вміст зсувного регістра пересилається у буфер приймача UDR, звідки прийняте значення має бути зчитаним.

Якщо формат кадру передбачає 9 біт даних, тоді перед записом в регістр UDR молодших 8 біт необхідно виставити у потрібне значення біт TXB8 (регістр UCSRB). Аналогічно і при прийомі даних, спершу необхідно прочитати значення біту RXB8 (регістр UCSRB), а потім вже читати значення молодших 8-ми бітів у регістрі UDR.

При прийомі даних також можемо виконати перевірку прапорів помилок (регістр UCSRA), які мають бути перевіреними ще перед читанням регістру даних UDR:

UPE – прапор помилки контролю парності, який виставляється при виявлені помилки парності у прийнятих даних.

DOR – прапор переповнення, який виставляється при виявленні нового старт-біта у зсувному регістрі, а буфер приймача у цей момент є заповнений (2 значення).

FE – прапор помилки кадрування, який виставляється при виявленні у прийнятому кадрі «0» на місці першого стоп-біта.

Для сповіщення про події: прийнято новий байт даних, завершено передачу даних, регістр даних UDR порожній передбачені відповідні прапорці RXC, TXC, UDRE (регістр UCSRA).

На основі цих прапорців також можуть бути згенеровані переривання для обробки цих подій. Дозвіл на переривання визначаються відповідними прапорами дозволів (регістр UCSRB):

RXCIE – дозвіл на переривання по завершенню прийому;

TXCIE – дозвіл на переривання по завершенню передачі;

UDRIE – дозвіл на переривання при спорожненні регістра UDR.

Переривання по завершенню передачі даних використовуються лише в окремих випадках. Наприклад, для переключення кінцевого пристрою у режим прийому по завершенню передачі даних в протоколі передачі даних RS-485.

Приклад. Реалізувати передачу даних через модуль UART із зовнішнім пристроєм. Згідно рис. 4.6 підключаються до МК дві кнопки для надсилання кодів символів ‘A’ та ‘B’, та три світлодіоди, які засвічуються при надсиланні на вхід модуля UART кодів символів ‘1’, ‘2’ та ‘3’. При прийомі інших значень світлодіоди гаснуть.

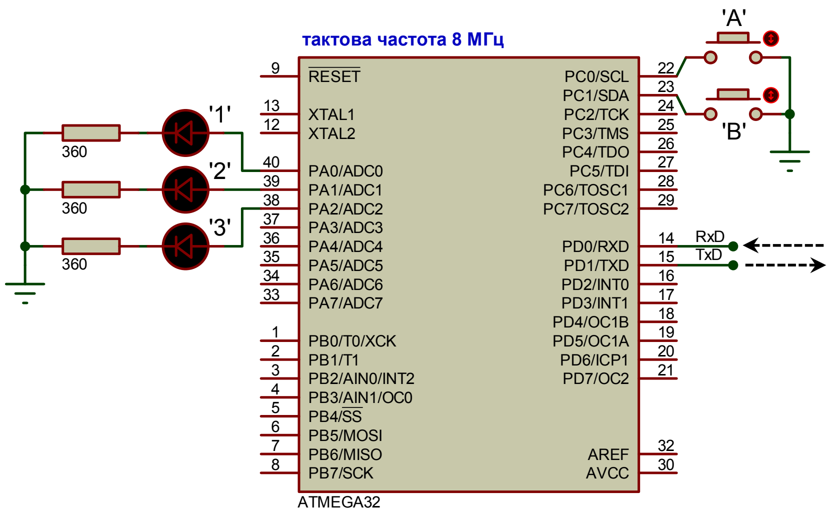


Рис. 4.6. Схема підключення МК для прийому/передачі даних по UART

У програмному коді прикладу буде задіяна лише підпрограма переривання при прийнятті байду даних. Макроси для підпрограм переривань при завершенні передачі та спорожненні буферу UDR наводяться для наочності, але не є реалізовані та не задіяні.

```
#include      <avr/io.h>
#include      <util/delay.h>
#include      <avr/interrupt.h>
#define       F_CPU        8000000L
#define       BAUD         9600
#define       UBRRcalc     (F_CPU/(BAUD*16L)-1)

ISR(USART_RXC_vect)              // переривання при прийнятому байті даних
{
        switch(UDR)
        {
                case '1':  PORTA = 1<<0;  break;
                case '2':  PORTA = 1<<1;  break;
                case '3':  PORTA = 1<<2;  break;
                default:   PORTA = 0;
        }
}

ISR(USART_TXC_vect )             // переривання при завершенні передачі
{       }
ISR(USART_UDRE_vect )            // переривання при спорожненні буферу UDR
{       }

int  main(void)
{
        Init();                 // ініціалізація периферії
        sei();                  // загальний дозвіл на переривання
        while (1)
   {
                if(bit_is_clear( PINC, 0) )          // якщо натиснута кнопка A
                { UDR = 'A';  _delay_ms(500); }      // передача 'A' + затримка

                if(bit_is_clear( PINC, 1) )          // якщо натиснута кнопка B
                { UDR = 'B';  _delay_ms(500); }      // передача 'B' + затримка
   }
}

void Init()
{
        DDRA=0xFF; PORTA=0x00;                      // на вихід
        DDRC=0x00; PORTC=0xFF;                      // на вхід
        // ініціалізація USART (асинхр. режим)
        // швидкість (регістр UBRR)
        UBRRL = (unsigned char)(UBRRcalc);
        UBRRH = (unsigned char)(UBRRcalc>>8);
        // формат кадру 8n2 без перевірки парності
        UCSRC = (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0)|(1<<USBS);
        UCSRA = 0;                 //скид прапорців регістра UCSRA
        // дозвіл прийому-передачі + перерив.прийому(регістр UCSRB)
        UCSRB = (1<<RXEN)|(1<<TXEN)|(1<<RXCIE);
}
```

4.10. Реалізація кільцевого буфера FIFO.

Обмін даними через модуль UART має суттєвий недолік: не можливо відправити на передачу одразу пакет байтів. Необхідно постійно очікувати спорожнення буфера UDR, а тоді вже відправляти черговий байт на передачу. Наприклад, для такого коду буде відправлено лише один-два байта:

```
for(int i=1; i<=10; i++)
        UDR =i;
```

Для спрощення організації обміну даними між різними пристроями чи процесами використовуються програмні кільцеві (циклічні) буфери FIFO (First Input First Output). Їх можна використовувати як для організації прийому/передачі модуля UART, так і, наприклад, для запису пакету даних в пам’ять EEPROM.

Кільцевий буфер є зручний тим, що дає можливість дуже просто виконувати заповнення буфера, перевірку наявності даних в буфері та вибірку даних з нього. При цьому не потрібно особливо турбуватися за границю буфера, переповнення пам’яті і т.п. Кільцевий буфер дозволяє коректно обмінюватися даними між обробником переривань та основною програмою.

Для реалізації кільцевого буфера необхідно в оперативній пам’яті виділити неперервний блок Buffer[BufSize] (рис. 4.7), розміром кратним степені 2 та два індекси StartB і EndB. Індекс StartB вказує на місце читання даних з буфера, а EndB на місце для запису у буфер. Розмір буфера вибирається кратним степені 2 для того, щоб було зручно маніпулювати цими індексами за допомогою інкремента та накладання маски за допомогою порозрядного AND. Наприклад, якщо розмір буфера дор. 16 (0b10000), тоді маска вибирається BufSize-1, тобто 15 (0b01111). Діапазон значень індексів для масиву з 16 елементів складає [0; 15]. Тому накладання цієї маски за допомогою порозрядного AND на значення індексів від 0 до 15 не змінює їх. А от якщо до максимального значення 15 додати 1, тобто вийде 16, та накласти нашу маску, то тоді значення обнулиться, та індекс буде вказувати на нульовий елемент масиву даних. При такому принципі роботи з індексом відбувається автоматичний перехід на початок буфера, як тільки ми досягнемо його кінця. Найоптимальніші 256-байтні буфери, так як у якості індексу можна використати цілий 1 байт, і тоді вже немає потреби накладати на них маску при інкременті їхніх значень.

Якщо буфер порожній, тоді індекси StartB та EndB мають однакове значення.

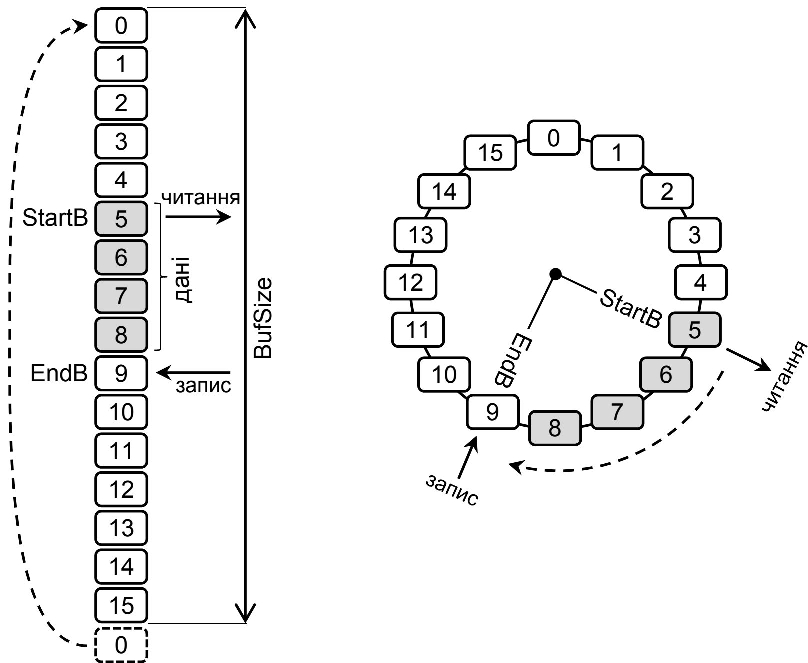


Рис. 4.7. Структура кільцевого буфера FIFO

Запис значення у буфер виконується за допомогою такого коду:

```
Buffer[EndB++] = value;
EndBuf &= BufMask;
```

Зчитування значення з буфера виконується аналогічно:

```
value = Buffer[StartB++];
StartB &= BufMask;
```

Про наявність даних у буфері свідчить відмінність значень індексів StartB та EndB:

```
while( StartB != EndB )    // поки буфер не порожній
{
// зчитуємо дані з буфера та опрацьовуємо їх
}
```

Для очистки буфера прирівнюємо між собою робочі індекси:

```
StartB = EndB;
```

Приклад. Реалізувати 2 кільцеві буфери для прийому та передачі даних через модуль UART. Згідно рис. 4.8, одна з кнопок має надсилати на вихід UART одразу 5 байтів даних. Періодично МК завантажується тривалою роботою, під час якої прийняті дані резервуються у вхідному буфері. По завершенню цієї роботи усі прийняті дані відсилаються назад, і МК знову переходить на виконання основної тривалої роботи. Два світлодіоди сигналізують про заповнення наших буферів, а окрема кнопка виконує скид помилки заповнення буферів.

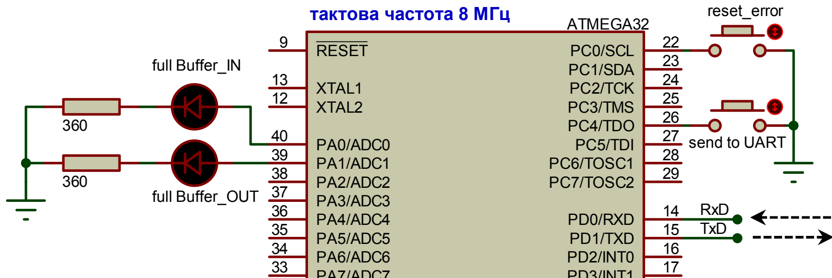


Рис. 4.8. Схема підключення МК для реалізації кільцевих буферів

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#define    F_CPU       8000000L
#define    BAUD        9600
#define    UBRRcalc    (F_CPU/(BAUD*16L)-1)

#define    BufSize     16
#define    BufMask     (BufSize - 1)

unsigned char  BufferIN[BufSize],  StartBufIN=0,  EndBufIN=0,  BufINer=0;
unsigned char  BufferOUT[BufSize], StartBufOUT=0, EndBufOUT=0,     BufOUTer=0;

//---------------------------------------------------------------------------------------------------------
void WriteBufOUT(unsigned char value)    // запис у буфер передачі
{
        BufferOUT[EndBufOUT++] = value;
        EndBufOUT &= BufMask;
        // перевіряємо на переповнення буфера
        if( StartBufOUT == EndBufOUT )
                BufOUTer = 1;                  // помилка: буфер заповнений
        // даємо дозвіл на переривання спорожнення UDR
        UCSRB |= 1<<UDRIE;
}
```

```
//---------------------------------------------------------------------------------------------------
unsigned char ReadBufIN(void)              // читання з буферу прийому
{
        unsigned char value = BufferIN[StartBufIN++];
        StartBufIN &= BufMask;
        return value;
}
//---------------------------------------------------------------------------------------------------
ISR(USART_RXC_vect)              // переривання при прийнятому байті даних
{       // запис у буфер прийому
        BufferIN[EndBufIN++] = UDR;
        EndBufIN &= BufMask;
        // перевіряємо на переповнення буфера
        if( StartBufIN == EndBufIN )
                BufINer = 1;                  // помилка: буфер заповнений
}
//---------------------------------------------------------------------------------------------------
ISR(USART_UDRE_vect )            // переривання при спорожненні буферу UDR
{
        UDR = BufferOUT[StartBufOUT++];
        StartBufOUT &= BufMask;
        // перевіряємо на спорожнення буфера передачі
        if( StartBufOUT == EndBufOUT )
                UCSRB &= ~(1<<UDRIE); // заборона на перерив. спорожн. UDR
}

int main(void)
{
        Init();                    // ініціалізація периферії
        sei();                     // загальний дозвіл на переривання
        while (1)
        {
                if( bit_is_clear(PINC, 0) )       // якщо натисн. кнопка «reset error»
                    { PORTA = 0; BufINer=0; BufOUTer=0; }

                if( bit_is_clear(PINC, 4) )       // якщо натисн. кнопка «send to UART»
                    for(int i=100; i<105; i++)
                            WriteBufOUT(i);   // запис у буфер передачі

                // тривала робота 2 сек.
                _delay_ms(2000);

                while( StartBufIN != EndBufIN )          //поки вх. буфер не порожній
                    WriteBufOUT( ReadBufIN() );          // BufOUT <- BufIN
                // індикація помилок
                if(BufINer)  PORTA=1<<0;                 // заповнення буфера BufIN
                if(BufOUTer) PORTA=1<<1;                 // заповнення буфера BufOUT
        }
}
```

```
void Init()      // ініціалізація периферії
{
        DDRA=0xFF; PORTA=0x00;                  // на вихід
        DDRC=0x00; PORTC=0xFF;                  // на вхід
        // ініціалізація USART (асинхр. режим)
        // швидкість (регістр UBRR)
        UBRRL = (unsigned char)(UBRRcalc);
        UBRRH = (unsigned char)(UBRRcalc>>8);
        // формат кадру 8n2 без перевірки парності
        UCSRC = (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0)|(1<<USBS);
        UCSRA = 0;              // скид прапорців регістра UCSRA
        // дозвіл прийому-передачі + перерив.прийому(регістр UCSRB)
        UCSRB = (1<<RXEN)|(1<<TXEN)|(1<<RXCIE);
}
```

Зауваження: після запису даних у буфер передачі необхідно надати дозвіл на переривання для події спорожнення буфера UDR. Аналогічно, при завершенні передачі останнього байта з вихідного буфера необхідно заборонити це переривання.

4.11. Інтерфейс RS-232.

Інтерфейс RS-232 був розроблений для забезпечення зв'язку між термінальним обладнанням та апаратурою передачі даних, використовуючи послідовний обмін двійковими даними.

Стандарт RS-232 був розроблений у 1969 році американською Асоціацією електронної промисловості, та після незначних поправок отримав назву RS-232C. У 1991 році була здійснена модифікація цього стандарту, після чого він отримав назву EIA/TIA-232E. Інша відома назва цього протоколу ITU V.24. Загально прийнятою назвою є EIA-232, або просто RS-232.

У стандарті передбачені асинхронний та синхронний режими обміну, однак переважна більшість пристроїв (наприклад ПК) працюють лише в асинхронному режимі.

Варто зазначити, що інтерфейс не забезпечує гальванічної розв'язки пристроїв.

Згідно інтерфейсу RS-232 (рис. 4,9), логічній «1» відповідає напруга на вході приймача в діапазоні від -12 до -3 В. Для ліній послідовних даних цей стан називається «MARK» (лог. «1»). Логічному «0» відповідає діапазон від +3 до +12 В. Для послідовних даних цей стан називається SPACE. Діапазон від -3 до +3 В – зона нечутливості, обумовлена гістерезисом приймача: стан ліній буде

рахуватися зміненим лише після перетинання порогу чутливості. Рівні сигналів на виходах передавачів повинні бути в діапазонах від -12 до -5 В та від +5 до +12 В для представлення, відповідно, «1» та «0». Різниця потенціалів між схемними землями з’єднувальних пристроїв повинна бути меншою 2 В, інакше можливе неправильне сприйняття сигналів.

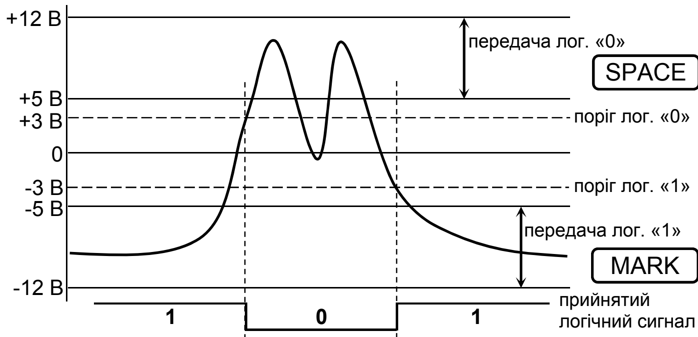


Рис. 4.9. Прийом сигналів RS-232

Формат кадру інтерфейсу RS-232 співпадає з форматом кадру модуля UART (п.4.9а, стр. 109). На рис. 4.10 зображено формат кадру для числа 82 у протоколі RS-232, що є подібним до представлення цього числа у TTL-рівнях модуля UART (рис. 4.3).

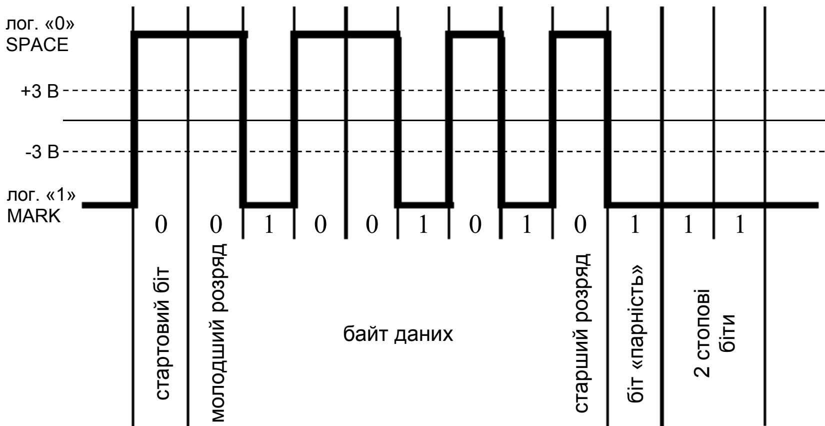


Рис. 4.10. Представлення числа 82 (01010010_2)
у кадрі 8n2 з перевіркою на парність у рівнях RS-232

Для підключення МК через модуль UART до COM-порту ПК необхідно виконати перетворення TTL-рівнів у рівні інтерфейсу RS-232 та навпаки за допомогою спеціалізованої мікросхеми, наприклад, MAX232.

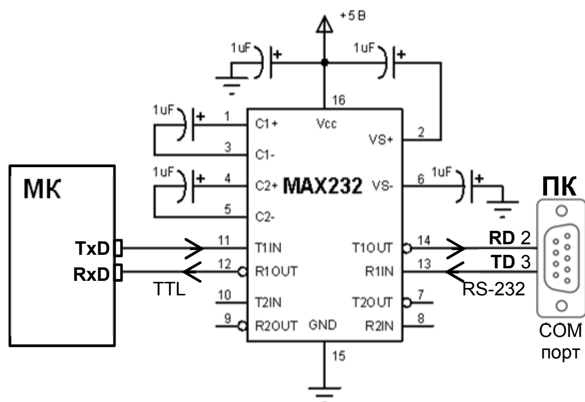


Рис. 4.10. Підключення модуля UART до ПК

Якщо на ПК відсутній фізичний COM-порт, тоді необхідно або використовувати плати розширення PCI з підтримкою RS-232 чи, скажімо, адаптери USB/RS-232, або спеціалізовані мікросхеми на зразок FT232RL.

4.12. Інтерфейс RS-485.

RS-485 (інша назва EIA/TIA-485) – найпоширеніший стандарт фізичного рівня зв’язку (канал зв’язку + спосіб передачі сигналу). Цей інтерфейс забезпечує обмін даними між декількома пристроями по одній двопровідній лінії зв’язку в напівдуплексному режимі. Для каналу зв’язку вибирається вита пара.

В основі інтерфейсу RS-485 лежить принцип диференціальної (балансної) передачі даних. По одному дроті (умовно А) іде оригінальний сигнал, а по іншому (В) – його інверсна копія. Тобто, якщо на одному дроті «1», то на іншому «0», і навпаки. Тому між двома дротами витої пари завжди є різниця потенціалів»: при логічній «1» вона позитивна, а при «0» – негативна (рис. 4.11). Такий спосіб передачі забезпечує високу стійкість до синфазних перешкод (що діють на два дроти лінії одночасно).

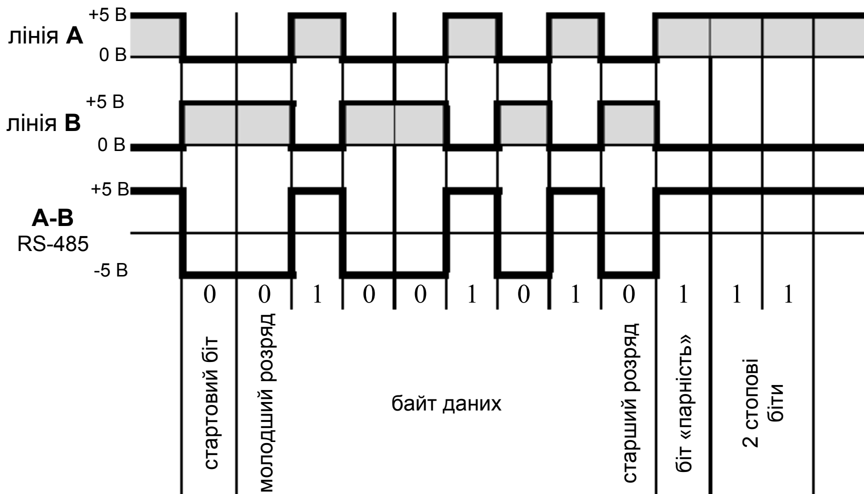


Рис. 4.11. Представлення числа 82 (01010010_2)
у кадрі 8n2 з перевіркою на парність у рівнях RS-485

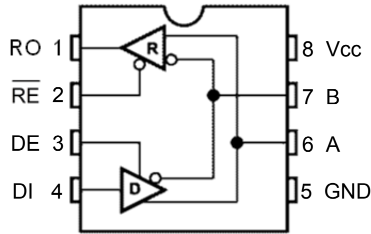
Стандарт RS-485 описує лише фізичний рівень процедури обміну даними. Його основні задачі це:

– перетворення вхідної послідовності «1» та «0» у диференціальний сигнал;
– передача диференціального сигналу в симетричну лінію зв'язку;
– підключення чи відключення передавача драйвера згідно сигналу верхнього протоколу обміну;
– прийом диференціального сигналу з лінії зв'язку.

Решта особливостей обміну, синхронізації та квітування покладається на верхній протокол обміну, наприклад RS-232 чи ModBus.

RS-485 забезпечує передачу даних зі швидкістю до 10 Мбіт/сек. Максимальна дальність залежить від швидкості: при швидкості 10 Мбіт/сек максимальна довжина лінії – 120 метрів, при швидкості 100 Кбіт/сек – 1200 метрів.

Апаратно інтерфейс реалізовується за допомогою спеціалізованих мікросхем прийомопередавачів з диференціальними входами/виходами (до лінії зв'язку) та цифровими портами (до портів UART МК), наприклад, MAX485(Maxim), ST485 (STMicroelectronics) (рис. 4.12).



Умовні позначання:
D (driver) – передавач
R (receiver) – приймач
RO (receiver output) – цифровий вихід приймача
$\overline{\text{RE}}$ (receiver output enable) – дозвіл роботи приймача
DE (driver output enable) – дозвіл роботи передавача
DI (driver input) – цифровий вхід передавача
A – прямий диференціальний вхід/вихід
B – інверсний диференціальний вхід/вихід

Рис. 4.12. Умовні позначення виводів MAX485, ST485

Цифровий вихід RO підключається до RxD приймача UART, а цифровий вхід DI до TxD передавача UART (рис. 4.13). Оскільки на диференціальній стороні приймач та передавач об’єднані, то на час приймання необхідно відключати передавач, а на час передачі – приймач. Для цього призначені керуючі входи $\overline{\text{RE}}$ (дозвіл приймача) та DE (дозвіл передавача). Вхід дозволу роботи приймача є інверсним, тому його можна об’єднати з входом дозволу роботи передавача, та одним сигналом з будь-якого порту МК перемикати між собою приймач та передавач. Якщо рівень «1» – працює передавач, якщо ж «0» – приймач.

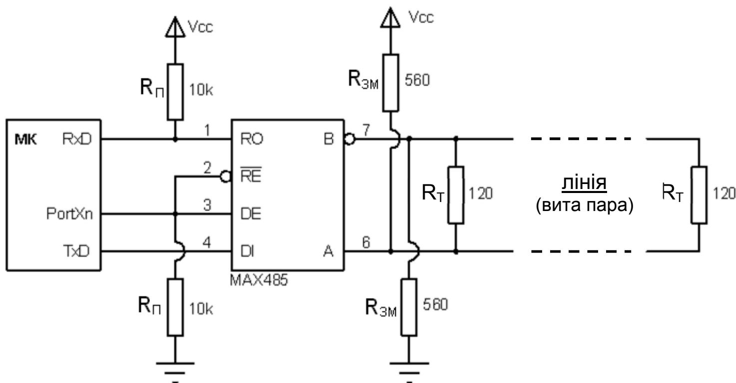


Рис. 4.13. Підключення прийомопередавача RS-485 до МК

Під час роботи прийомопередавача RS-485 на передачу вихід приймача RO переводиться у третій стан і вивід RxD МК «повисає у повітрі». Тому, замість високого рівня «1» відсутності передачі, будь-яка завада може бути прийнята за вхідний сигнал. Для цього необхідно на час передачі відключати приймач UART (через керуючий регістр), або підтягувати вивід RxD за допомогою резистора до «1». Також можна на МК задіяти внутрішній підтягуючий резистор.

При подачі живлення на МК пройде певний час, поки вивід керування дозволами роботи драйвера RS-485 буде проініціалізований на вихід. До цього моменту він буде функціонувати як високоімпедансний вхід. Тому існує можливість, що якоюсь завадою буде активований передавач RS-485, і у лінію буде відправлене «сміття». Щоб уникнути цього, рекомендується виводи керування роботою драйвера підтягнути резистором до «землі».

Чутливість приймача RS-485 може бути різною, але гарантований граничний діапазон розпізнавання сигналу становить ±200 мВ. Тобто, коли U_{AB}>+200 мВ – приймач визначає «1», якщо U_{AB}<-200 мВ, тоді приймач визначає «0». Якщо різниця потенціалів є меншою за визначені допустимі границі, тоді правильне розпізнавання сигналу не гарантується. Таке може трапитися або при від'єднанні приймача від лінії, або при відсутності в лінії активних передавачів, коли ніхто не задає рівень. Щоб уникнути видачі помилкових сигналів на вхід UART, необхідно на входах A-B гарантувати різницю потенціалів U_{AB}>+200 мВ. Цей зсув, при відсутності вхідних сигналів, забезпечує на виході приймача лог. «1», підтримуючи стан відсутності передачі. Для цього прямий вхід A необхідно підтягнути резистором до живлення, а інверсний вхід B до «землі» (рис. 4.13).

Для уникнення ефекту довгої лінії, коли сигнали мають здатність відбиватися від відкритих кінців лінії передачі та її відгалужень, необхідно на віддалених кінцях лінії між провідниками витої пари включати узгоджувальні резистори (термінатори). Номінал резисторів має бути рівним хвильовому опорі лінії передачі. Як правило він становить 120 Ом. Для коротких ліній (кілька десятків метрів) та при низьких швидкостях (менше 38 400 біт/сек) узгодження можна не робити взагалі.

Лінія зв'язку повинна представляти собою один кабель витої пари. До цього кабелю приєднуються всі прийомопередавачі RS-485. Відстань від лінії до драйверів RS-485 повинна бути як можна коротшою, оскільки довгі відгалуження вносять неузгодженість та викликають відбиття.

Це має і свій великий недолік: підвищене споживання струму від передавача, оскільки в лінію включається низькоомне навантаження, а також зменшене число підключених прийомпередавачів RS-485. Тому можна використати більш економне схемне рішення (рис. 4.14).

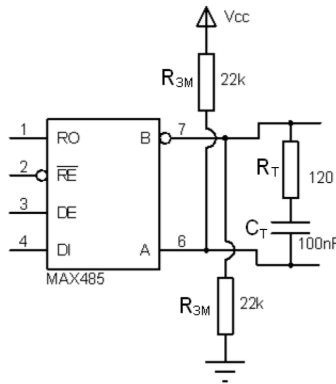


Рис. 4.14. Економне підключення прийомопередавача RS-485 до МК

Для нормального функціонування мережі RS-485 необхідно, щоб сигнальні «землі» пристроїв були з'єднані між собою. Рекомендується таке з'єднання виконувати за допомогою резистора 100 Ом для кожного драйвера (рис. 4.15). Сигнальні «землі» та захисні заземлення мають також бути з'єднані між собою резистором 100 Ом.

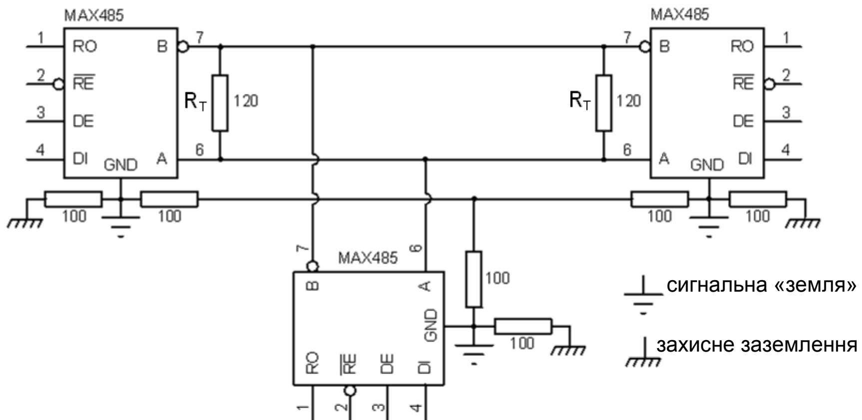


Рис. 4.15. Реалізація кіл сигнального заземлення

Згідно зі специфікацією RS-485, на лінії можуть знаходитися до 32 прийомопередавачів, враховуючи узгоджуючі резистори (120 Ом). Це обумовлено вхідним опором приймача 12 КОм з боку лінії. Деякі мікросхеми драйверів мають підвищений вхідний опір, і тому дають можливість підключати до лінії більшу кількість пристроїв.

4.13. Мультипроцесорний режим модуля UART.

Цей режим дає можливість ефективно організувати обмін даними між основним (master) та рядом підлеглих (slave) МК по протоколу RS-485. При цьому, кожному підлеглому призначається своя унікальна адреса, згідно якої основний МК буде звертатися до нього.

Для цього необхідно налаштувати модулі UART усіх МК у мережі RS-485 на 9-ти бітний обмін даними та усі підлеглі МК переключити у режим мультипроцесорного обміну, встановивши в «1» розряд MPCM у регістрі керування UCSRA.

Поведінка підлеглих МК з активованим мультипроцесорним режимом залежить від значення старшого розряду даних. Якщо він встановлений в «1», то це означає, що основний МК надсилає адресу, і тоді усі підлеглі приймають цей адресний байт даних та порівнюють зі значенням своєї адреси. Якщо для якогось з них адреси співпадають, тоді цей підлеглий МК відключає для себе мультипроцесорний режим та переходить у звичайний режим прийому даних. Решта МК і далі знаходяться в мультипроцесорному режимі. Якщо основний МК надсилає дані зі скинутим в «0» старшим розрядом, то ці дані може прийняти лише той МК, який працює у звичайному режимі, тобто для якого попередня адреса була співпала з його власною. Решта підлеглі МК ігнорують ці дані, і завдяки цьому зменшується процесорне навантаження на них. По завершенню обміну даними з «базою» вибраний підлеглий МК знову активує мультипроцесорний режим та переходить у режим фільтрації даних.

Приклад. Реалізувати обмін даними по протоколу RS-485 між базою (master) та двома клієнтами (slave). Адреса кожного клієнта задається за допомогою 8-ми клавішного перемикача типу «піаніно», що підключене до порту C. Кожен клієнт також має підключені до портів A та B 4-клавішні «піаніна», які умовно позначені як Left та Right. База має 3 кнопки: перша вибирає клієнта A чи B, друга визначає які значення (Left чи Right) необхідно надіслати клієнтові для бази, а третя, згідно вибраних параметрів за допомогою попередніх кнопок, надсилає запит вибраному клієнту. Вибір клієнта, піаніна та отриманого значення від клієнта відображається на індикації (рис. 4.16).

* Підтягувальні резистори для драйверів MAX485
та термінальні резистори на схемі не вказані

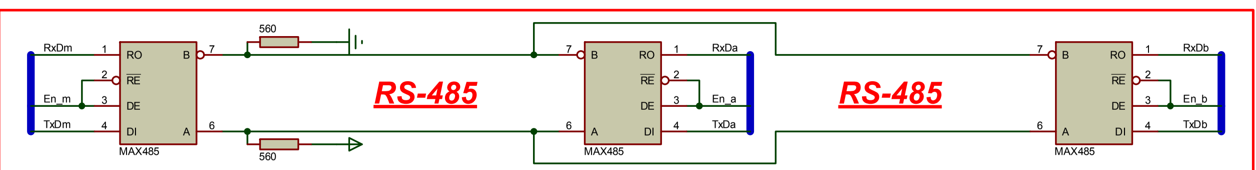


Рис. 4.16. Принципова схема для дослідження мультипроцесорного режиму модуля UART

Програмний код МК «master»

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#define    F_CPU          8000000L
#define    BAUD           9600
#define    UBRRcalc       (F_CPU/(BAUD*16L)-1)
#define    BUF_SIZE       16
#define    BUF_MASK       (BUF_SIZE-1)

unsigned char  BufferOUT[BUF_SIZE][2],  StartBufOUT = 0, EndBufOUT = 0;
volatile unsigned char  waitread = 0;         //

// -------------- запис у буфер передачі ----------------------------------------------------
void WriteBufOUT(unsigned char value, unsigned char bit8)
{
        BufferOUT[EndBufOUT][0] = value;
        BufferOUT[EndBufOUT++][1] = bit8;
        EndBufOUT &= BUF_MASK;
        cli();                                  // загальна заборона на переривання
        if( waitread == 0 )
                UCSRB |= 1<<UDRIE; // дозвіл на переривання спорожнення UDR
        sei();                                  // загальний дозвіл на переривання
}
 // -------------- переривання при прийнятому байті даних ------------------------------
ISR(USART_RXC_vect)
{
        PORTB = UDR;
        waitread = 0;
        // перевіряємо на спорожнення буфера передачі
        if( StartBufOUT != EndBufOUT )
                UCSRB |= 1<<UDRIE; // дозвіл на переривання спорожнення UDR
}
// -------------- переривання при завершенні передачі -------------------------------------
ISR(USART_TXC_vect )
{ PORTD &= ~(1<<PD2); }            //режим прийому
// -------------- переривання при спорожненні буферу UDR ------------------------------
ISR(USART_UDRE_vect )
{
        PORTD |= 1<<PD2;                                //режим передачі
        if( BufferOUT[StartBufOUT][1] == 1 )   UCSRB |= 1<<TXB8;      //bit8=1
        else     { UCSRB &= ~(1<<TXB8); waitread = 1; }       //bit8=0; очік. прийому
        asm("nop");                                     // затримка в 1 такт
        UDR = BufferOUT[StartBufOUT++][0];              // читання з FIFO-буфера
        StartBufOUT &= BUF_MASK;
        // перевіряємо на спорожнення буфера передачі
        if( StartBufOUT == EndBufOUT  ||  waitread == 1 )
                UCSRB &= ~(1<<UDRIE); // заборона на перер. спорожнення UDR
}
```

```
int main(void)
{
        Init();                 // ініціалізація периферії
        sei();                  // загальний дозвіл на переривання
        while (1)
        {
            if( bit_is_clear(PINC, 0) )         // надіслати запит
                {
                    if( bit_is_set(PINA, 0) )   WriteBufOUT('A', 1);    // адреса A
                    else                        WriteBufOUT('B', 1);    // адреса B
                    if( bit_is_set(PINA, 6) )   WriteBufOUT('L', 0);    // лівий
                    else                        WriteBufOUT('R', 0);    // правий
                    _delay_ms(500);             // затримка 0.5 сек.
                }

            if( bit_is_clear(PINC, 1) )         // вибір підлеглого
                {
                    if( bit_is_set(PINA, 0) )   {PORTA &= ~(1<<0); PORTA |= 1<<1;}
                    else                        {PORTA &= ~(1<<1); PORTA |= 1<<0;}
                    _delay_ms(500);             // затримка 0.5 сек.
                }

            if( bit_is_clear(PINC, 2) )         // з якого піаніна надіслати дані
                {
                    if( bit_is_set(PINA, 6) )   {PORTA &= ~(1<<6); PORTA |= 1<<7;}
                    else                        {PORTA &= ~(1<<7); PORTA |= 1<<6;}
                    _delay_ms(500);             // затримка 0.5 сек.
                }
    }
}

void Init()     // ініціалізація периферії
{
        DDRA=0xFF; PORTA=0x00;                      // на вихід - 0В
        DDRB=0xFF; PORTB=0x00;                      // на вихід - 0В
        DDRC=0x00; PORTC=0xFF;                      // на вхід  - Rпідт.
        DDRD=0b11111110; PORTD=0b00000001;          // pin0 - Rпідт
        // ініціалізація USART (асинхр. режим)
        // швидкість (регістр UBRR)
        UBRRL = (unsigned char)(UBRRcalc);
        UBRRH = (unsigned char)(UBRRcalc>>8);
        //обнулення регістра UCSRA
        UCSRA = 0;
        // формат кадру 9n2 без перевірки парності
        UCSRC = (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0)|(1<<USBS);
        // дозвіл прийому-передачі+перерив.прийому, заверш.перед.+ 9n
        UCSRB = (1<<UCSZ2)|(1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE);
}
```

Пояснення до програми. Для організації передачі даних через модуль UART використовуємо кільцевий 16-елементний буфер розмірністю 16×2. Один байт для 8 біт даних, другий для старшого розряду 9-бітної посилки, який буде вказувати чи посилка адресна (1), чи містить дані (0).

База надсилає дві посилки (адреса+дані) та очікує відповіді від вибраного клієнта. За допомогою змінної waitread організовується заборона на пересилання даних з кільцевого буфера до передавача UART до моменту відповіді від клієнта.

Оскільки на принциповій схемі ми не передбачили зовнішніх підтягуючих резисторів до високого рівня входів RxD модулів UART, то виводи RxD конфігуруються на внутрішній підтягуючий резистор.

Термінальні резистори 120 Ом також відсутні на принциповій схемі (через них пакет Proteus не коректно працював).

Чотири світлодіоди призначені для індикації вибраного клієнта (A чи B) та вказаного піаніна (Left чи Right), значення якого має бути надісланим від клієнта. На стовпцевий індикатор з 4-х сегментів, що підключений до порту B, виводиться значення, отримане від клієнта.

Програмний код МК «slave»

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#define         F_CPU       8000000L
#define         BAUD        9600
#define         UBRRcalc    (F_CPU/(BAUD*16L)-1)

unsigned char Address;
// -------------- переривання при прийнятому байті даних -------------------------------
ISR(USART_RXC_vect)
{
        if( bit_is_set(UCSRA, MPCM) )
        {
                if( UDR == Address )
                    UCSRA &= ~(1<<MPCM);            //відключ.мультипроц.режим
        }
        else
        {
                PORTD |= 1<<PD2;                    // режим передачі
                if( UDR == 'L' )        UDR = ~PINA;    // надсил. знач. Left
                else                    UDR = ~PINB;    // надсил. знач. Right
                UCSRA |= (1<<MPCM);                 // включ.мультипроц.режим
        }
}
```

```
// -------------- переривання при завершенні передачі ------------------------------------
ISR(USART_TXC_vect )
{  PORTD &= ~(1<<PD2); }                        //режим прийому

int main(void)
{
	Init();                                     // ініціалізація периферії
	sei();                                      // загальний дозвіл на переривання
	while (1)
	{     }
}

void Init()      // ініціалізація периферії
{
	DDRA=0x00; PORTA=0xFF;                      // на вхід  - Rпідт.
	DDRB=0x00; PORTB=0xFF;                      // на вхід  - Rпідт.
	DDRC=0x00; PORTC=0xFF;                      // на вхід  - Rпідт.
	DDRD=0b11111110; PORTD=0b00000001;   // pin0 - Rпідт
	//ініціалізація USART (асинхр. режим)
	//швидкість (регістр UBRR)
	UBRRL = (unsigned char)(UBRRcalc);
	UBRRH = (unsigned char)(UBRRcalc>>8);
	// мультипроцесорний режим
	UCSRA = (1<<MPCM);
	//формат кадру 9n2 без перевірки парності
	UCSRC = (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0)|(1<<USBS);
	//дозвіл прийому-передачі+перерив.прийому,заверш.перед.+9n
	UCSRB = (1<<UCSZ2)|(1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE);
	// читання адреси МК з піаніна
	Address = ~PINC;
}
```

Пояснення до програми. Значення, що зчитуються з перемикачів типу піаніно, інвертуються для отримання нормального вигляду, оскільки їхні виводи під'єднані до «землі». Адреса клієнта зчитується на етапі ініціалізації, і тому її нове значення буде задіяне лише після рестарту МК.

4.14. Аналогово-цифровий перетворювач (АЦП).

АЦП (analog-to-digital converter, ADC або A/D) дає еквівалентне представлення аналогового сигналу у цифровому (двійковому) коді. Це дає можливість МК працювати з аналоговими пристроями, наприклад, знімати покази з різноманітних давачів, що мають аналоговий вихід, виконувати контроль за рівнем напруги живлення, контролювати робочий струм виконавчих пристроїв тощо.

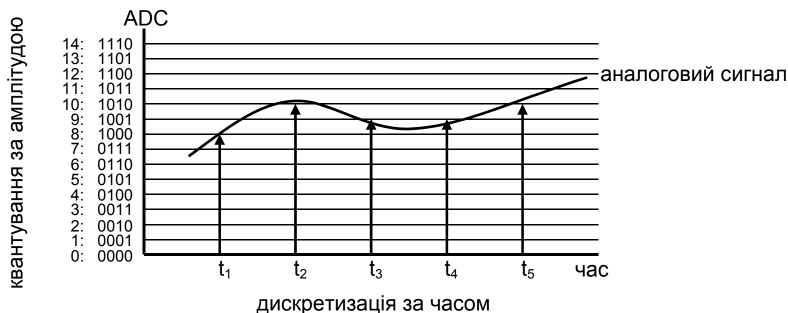


Рис. 4.17. Оцифрування аналогового сигналу

Основними параметрами будь-якого АЦП є розрядність та швидкодія, тобто максимальна частота дискретизації (вибірок за сек.).

Розрядність АЦП визначає кількість рівнів квантування, якими перетворювач може представити аналоговий сигнал. 8-ми розрядний АЦП забезпечує 2^8=256 рівнів, тобто діапазон значень на виході перетворювача 0…255; 10-розрядний – 10^8=1024 рівнів (значення від 0 до 1023). Якщо розрядність АЦП становить 10 біт, діапазон вхідної напруги від 0 до 5 В, тоді розрядність за напругою: (5-0)/1024=~4,88 мВ.

Частота дискретизації визначає, як часто ми можемо здійснювати вибірки цифрових значень з аналогового сигналу.

Швидкодія АЦП та його розрядність визначаються типом архітектури. Наприклад, паралельні АЦП можуть мати розрядність 8-12 біт та частоту дискретизації від 10 МГц до понад 1 ГГц; АЦП послідовного наближення – 10-16 біт та швидкодію від 100кГц до понад 1 МГц; сігма-дельта АЦП – 16-24 біт та швидкодію від 1 до 100 кГц; інтегруючі АЦП – 16-24 біт та швидкодію від 10 до 200 Гц. Розрядність та швидкодія взаємопов’язані параметри: більша розрядність – нижча швидкодія, і навпаки.

Деякі МК AVR мають інтегровані модулі АЦП послідовного наближення. Спрощена схема АЦП послідовних наближень представлена на рис. 4.18. Перетворювач складається з ЦАП (цифро-аналогового перетворювача), регістра послідовних наближень, компаратора та блоку синхронізації. У початковому стані у всіх розрядах (D_7, … , D_0) регістра послідовних наближень встановлені значення «0». Цикл вимірювання починається з того, що старший розряд D_7 регістру встановлюється в «1». Після цього за допомогою компаратора порівнюється напруга на виході ЦАП та на вході АЦП. Якщо напруга на вході виявляється більшою, тоді значення логічного сигналу у розряді D_7

зберігається рівним «1». У іншому випадку, старший розряд регістра обнулюється. Після цього розряд D_6 встановлюється в «1», та знову проводиться порівняння напруг. Цей цикл операцій послідовно застосовується до всіх решта розрядів регістра. Робота схеми синхронізується за допомогою сигналів початку та закінчення перетворення.

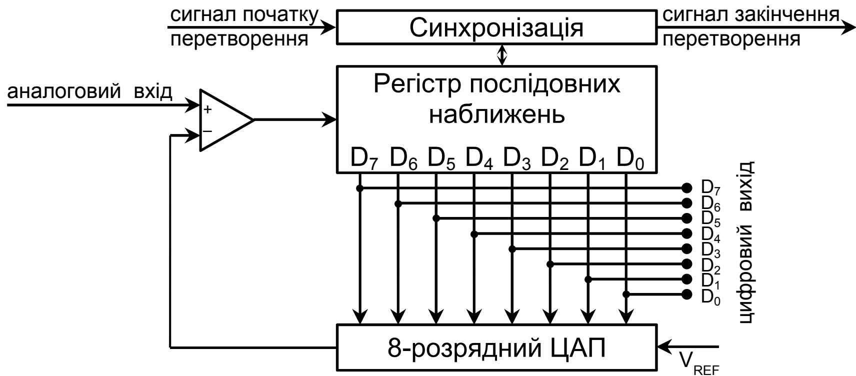


Рис. 4.18. Принцип дії АЦП послідовних наближень

Інтегрований модуль АЦП МК AVR має розрядність 10 біт, швидкодію до 15 тис. вибірок за секунду, інтегральну нелінійність 0,5 LSB (least significant bit, молодший біт), абсолютну похибку 2 LSB.

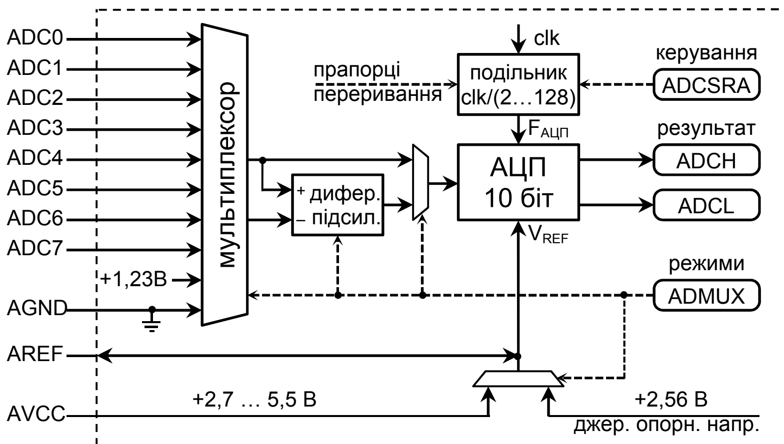


Рис. 4.19. Структурна схема модуля АЦП

Для роботи 10-розрядного АЦП необхідно 3 сигнали: вхідний аналоговий сигнал, тактовий $F_{АЦП}$ та взірцевий V_{REF}. Вхідний аналоговий сигнал надходить від мультиплексора, що виконує комутацію з 8-ми аналогових каналів ADC0…ADC7 та двох тестових напруг 0 та +1,23 В. Аналогові входи АЦП можуть бути використані, як 8 окремих каналів з несиметричними входами (сигнали відносно «землі»), так і можуть бути об'єднані попарно для формування каналів із диференціальними входами (до 13 різних варіантів). При цьому 2 канали мають можливість 20- та 200-кратного підсилення вхідного сигналу. При коеф. підсилення 1х та 20х роздільна здатність складає лише 8 біт, а при 200х – 7 біт. Робота з диференціальними каналами у МК в DIP-корпусі виробником не передбачена.

Як **джерело опорної напруги** для АЦП можуть використовуватися напруга живлення МК, внутрішня опорна напруга 2,56 В або зовнішнє джерело напруги, що подається на вивід AREF. Вибір джерела вказується у регістрі ADMUX (табл. 4.8). Оскільки вивід AREF має безпосередній електричний зв'язок з модулем АЦП, то, якщо на нього не подається зовнішня опорна напруга, рекомендується шунтувати вивід керамічним конденсатором 0,1 мкФ. Зовнішня опорна напруга не може бути вищою за напругу живлення МК.

Таблиця 4.8. Вибір джерела опорної напруги

REFS1	**REFS0**	**Джерело опорної напруги**
0	0	Зовнішнє джерело, підключене до AREF
0	1	Напруга живлення V_{CC}
1	1	Внутрішня опорна напруга 2,56 В

Тактовий частота АЦП $F_{АЦП}$ формується з тактового сигналу МК clk через окремий подільник, шляхом поділу його на коеф. 2…128, що задається в регістрі ADCSRA.

Таблиця 4.9. Вибір коефіцієнта подільника АЦП

	2	2	4	8	16	32	64	128
ADPS0	0	1	0	1	0	1	0	1
ADPS1	0	0	1	1	0	0	1	1
ADPS2	0	0	0	0	1	1	1	1

Найвища точність перетворення досягається при тактовій частоті модуля АЦП 50 … 200 кГц. Тому необхідно вибирати коефіцієнт поділу, щоб частота АЦП лежала в цьому діапазоні. На вищих частотах точність буде меншою.

Вибір аналогового каналу здійснюється за допомогою розрядів MUX4:0 регістра ADMUX. У табл. 4.10 наведено значення цих розрядів лише для вибору несиметричних входів та тестових напруг (для диференціальних каналів див. інструкцію до МК).

Таблиця 4.10. Вибір каналів АЦП з несиметричними входами

	ADC0	ADC1	ADC2	ADC3	ADC4
MUX4:0	00000	00001	00010	00011	00100
	ADC5	**ADC6**	**ADC7**	**1,22 В**	**0 В**
MUX4:0	00101	00110	00111	11110	11111

Якщо виконати переключення на інший канал під час перетворення, то зміна відбудеться лише після закінчення перетворення.

Модуль АЦП може виконувати перетворення у 2-х режимах:

- одинарне перетворення – запуск кожного перетворення виконується індивідуально користувачем чи програмою;
- неперервні перетворення – запуск перетворень здійснюється апаратно через визначені інтервали часу.

Режим перетворення визначається бітом ADATE регістру ADCSRA. Якщо він дорівнює «0», тоді модуль АЦП функціонує в режимі одинарного перетворення, а якщо «1», то запуск перетворення визначається значеннями бітів ADTS2:0, згідно табл. 4.11.

Таблиця 4.11. Вибір джерела автоматичного запуску АЦП

ADTS2	ADTS1	ADTS0	Джерело запуску АЦП
0	0	0	Автономний режим
0	0	1	Переривання від аналогового компаратора
0	1	0	Зовнішнє переривання INT0
0	1	1	Переривання таймера T0 по співпадінню
1	0	0	Переривання таймера T0 по переповненню
1	0	1	Переривання таймера T1 по співпадінню 'B'
1	1	0	Переривання таймера T1 по переповненню
1	1	1	Переривання таймера T1 згідно події capture

Запуск кожного перетворення в режимі одинарного перетворення, а також запуск першого перетворення в режимі неперервного перетворення здійснюється записом «1» у розряд ADSC регістра ADCSRA. Тривалість циклу перетворення складає 13-14 тактів МК. В автономному неперервному режимі новий цикл починається одразу після запису результату перетворення у регістри ADCH:ADCL.

Дозвіл роботи модуля АЦП надається при встановленні в «1» розряду ADEN регістра ADCSRA. **Дозвіл на переривання** АЦП надається при встановленні в «1» розряду ADIE регістра ADCSRA.

Після завершення перетворення **результат** записується у 2-х регістрах ADCH:ADCL. За замовчуванням (розряд ADLAR регістра ADMUX дор. «0») результат перетворення вирівнюється вправо:

-	-	-	-	-	-	bit9	bit8	**ADCH**
bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	**ADCL**

при значенні розряду ADLAR «1» результат вирівнюється вліво:

bit9	bit8	bit7	bit6	bit5	bit4	bit3	bit2	**ADCH**
bit1	bit0							**ADCL**

Зауваження: спершу необхідно прочитати значення регістра ADCL, а після цього вже ADCH. Це пов'язано з тим, що після звертання до ADCL процесор блокує для АЦП доступ до того часу, поки не буде зчитане значення з ADCH. Якщо ж нам достатньо точності 8-бітного значення, то можемо виконати вирівнювання вліво та зчитувати значення з регістра ADCH.

Результат перетворення АЦП для каналів з несиметричним входом визначається за такою формулою:

$$ADC = \frac{V_{IN} \cdot 1024}{V_{REF}}$$

для каналів з диференціальним входом:

$$ADC = \frac{(V_{POS} - V_{NEG}) \cdot K_{\text{підс}} \cdot 512}{V_{REF}}$$

де V_{IN} – значення вхідної напруги, V_{POS} – значення напруги на додатному вході, V_{NEG} – значення напруги на від'ємному вході, $K_{\text{підс}}$ – коеф. підсилення, V_{REF} – величина опорної напруги.

Для зменшення впливу завад від роботи процесора МК передбачений спеціальний **«сплячий» режим** ADC Noise Reduction. У цьому режимі з периферійних пристрої працюють лише: АЦП, зовнішні переривання, модуль розпізнавання адреси модуля TWI, таймер Т2 та сторожовий таймер. Для використання цього режиму необхідно встановити модуль АЦП в режим одинарного перетворення, виставити «сплячий» режим АЦП у регістрі MCUCR за допомогою бітів SM2:0 = 001 та у цьому ж регістрі встановити в «1» біт дозволу

переходу у сплячий режим SE. Після запуску команди SLEEP (зупинка процесора) почнеться цикл перетворення. По завершенню перетворення буде згенероване переривання від модуля АЦП, яке виведе МК у робочий режим, та почнеться виконання підпрограми оброблення цього переривання.

Зауваження 1: оскільки канали ADC0…ADC7 є також і входами PA0…PA7, то на етапі ініціалізації периферії необхідно відключити внутрішні підтягуючі резистори на виводах, до яких будуть подаватися аналогові сигнали. Також варто пам’ятати, що ці виводи мають в наявності захисні резистори, які обмежують зверху та знизу вимірювану напругу. Усе, що виходить за межі 0…V_{CC}, не обробляється та прирівнюється знизу до AGND та зверху до V_{CC}.

Зауваження 2: модуль АЦП є оптимізований для роботи з аналоговими сигналами з вихідним опором приблизно 10 кОм чи менше.

Приклад. На основі модуля АЦП (рис. 4.20) виконувати вимірювання напруги на змінному резисторі, підключеному до V_{CC}=5 В. Результати вимірювання виводити на LCD-дисплей (дивись лабораторну роботу №4).

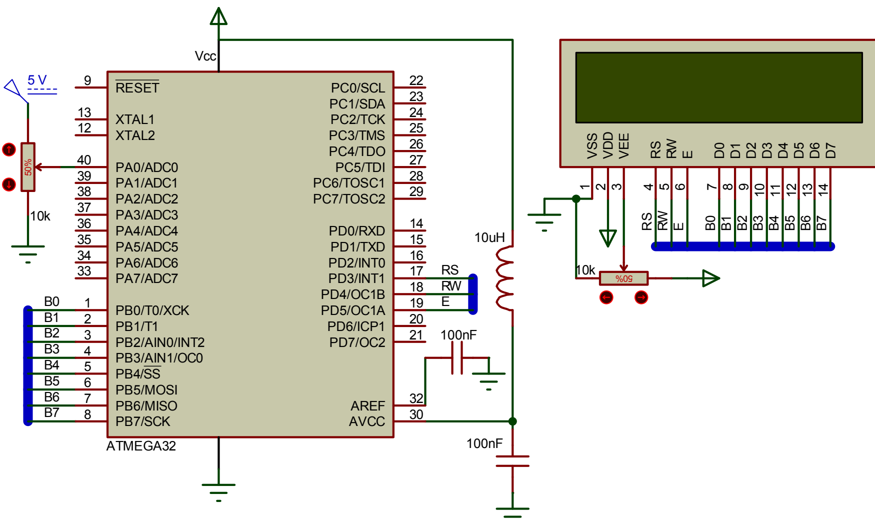


Рис. 4.20. Принципова схема дослідження роботи АЦП

* *Для зменшення шумів вивід AVCC повинен бути підключений до V_{CC} через LC-фільтр (рис. 4.20).*

Значення напруги на резисторі обчислюється за формулою:

$$V_{IN} = \frac{ADC \cdot 5\text{ В}}{1024}$$

Оскільки МК працює на частоті 8 МГц, то вибираємо подільник модуля АЦП рівним 64 (8 МГц / 64 = 125 кГц).

```
//=========== LCD Define ==========================================
#define    LCDdataPORT       PORTB      // LCD Data Port
#define    LCDdataPIN        PINB
#define    LCDdataDDR        DDRB

#define    LCDcontrolPORT    PORTD      // LCD Control Port
#define    LCDcontrolPIN     PIND
#define    LCDcontrolDDR     DDRD

#define    RS     3
#define    RW     4
#define    E      5
//=================================================================
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <math.h>
#include <stdlib.h>
#include "LCD_8.h"

//-------------Вектори переривань------------------------------------------
ISR(TIMER1_COMPA_vect)          // переривання таймера
{
        ADCSRA |= 1<<ADSC;
}
ISR(ADC_vect)                   // переривання АЦП
{
        char Sbuf[6];
        float Vin = ADC*5.0/1024;
        LCD_WriteCommand(_ClearDisplay);
        LCD_WriteStr( dtostrf(Vin,5,2,Sbuf) );
        LCD_WriteLetter(' ');
        LCD_WriteLetter('V');
}

void main()                     // основна програма
{
        Initializer();
        sei();
```

```
	while (1)			// основний робочий цикл
	{   }
}

void Initializer()			// Ініціалізація заліза
{
	//Ініціалізація портів
	//Порт A на вхід (0біт - АЦП)
	DDRA  = 0x00;
	PORTA = 0b11111110;
	//Порт B на вихід
	DDRB  = 0x00;
	PORTB = 0xFF;
	//Порт D на вихід
	DDRD  = 0xFF;
	PORTD = 0x00;
	//Ініціалізація LCD
	InitLCD();
	//Ініціалізація Таймера#1
	//Режим: Скид При Співпадінні OCR1A (1sec) + дільник=256
	OCR1AH = 0x7A;
	OCR1AL = 0x11;
	TCCR1A = 0x00;
	TCCR1B = (1<<WGM12) | (1<<CS02);
	//дозвіл на переривання по співпадінню
	TIMSK |= 1<<OCIE1A;
	//Ініціалізація АЦП=Vcc(~5V)  + right + ADC0 + Кпод=64
	ADMUX  = 1<<REFS0;
	ADCSRA = (1<<ADEN)|(1<<ADIE)|(1<<ADPS2)|(1<<ADPS1);
}
//-----------------------------------------------------------------------
```

Для роботи у «сплячому» режимі АЦП програму необхідно дещо модифікувати. Найперше необхідно додати у розділ макровизначень підключення бібліотеки

```
#include <avr/sleep.h>
```

Далі у розділі ініціалізації периферії (функція Initializer) додати код для визначення режиму ADC Noise Reduction та дозвіл на перехід у сплячий режим АЦП.

```
MCUCR = (1<<SE)|(1<<SM0);
```

Підпрограма оброблення переривання таймера та основна функція повинні виглядати таким чином.

```
char volatile timer=0;
ISR(TIMER1_COMPA_vect)
{
	timer = 1;
}

void main()
{
	Initializer();
	sei();
	while (1)			//основний робочий цикл
	{
		if( timer == 1 )
		{
		   timer = 0;
		   sleep_cpu();
		}
	}
}
```

Зауваження. Якщо виконувати перехід у сплячий режим у підпрограмі переривання таймера, то тоді програма функціонує невірно.

Перелік рекомендованої літератури

1. ATmega32A. Datasheets – Atmel Corporation. – 353 pages.
2. AVR204: BCD Arithmetics. Application Note – Atmel Corporation. – 14 pages.
3. AVR134: Real Time Clock (RTC) using the Asynchronous Timer. Application Note – Atmel Corporation. – 9 pages.
4. AVR242: 8-bit Microcontroller Multiplexing LED Drive and a 4x4 Keypad. Application Note – Atmel Corporation. – 26 pages.
5. Роман Абраш. Книга по работе с WinAVR и AVR Studio. – Підбірка статей у журналі «Радиолюбитель» за період 01/2010–05/2011. – 88 стр.
6. AVR Libc Development Pages. – http://www.nongnu.org/avr-libc.
7. Anil Kumar Maini. Digital Electronics: Principles, Devices and Applications. – John Wiley & Sons, Ltd. – 2007. – 727 pages.

Навчальне видання

Програмування мікроконтролерів систем автоматики: конспект лекцій для студентів базового напряму 050201 “Системна інженерія”.

Укладач: Павельчак Андрій Геннадійович
Самотий Володимир Васильович
Яцук Юрій Васильович