КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

В.М. Прокопець

ПРОГРАМУВАННЯ МІКРОКОНТРОЛЕРІВ AVR НА МОВІ С.

ЛАБОРАТОРНИЙ ПРАКТИКУМ.

НАВЧАЛЬНИЙ ПОСІБНИК

Київ 2018

Зміст

Лабораторна робота 0.
Знайомство з програмними та апаратними засобами програмування мікроконтролерів AVR 4

Лабораторна робота 1.
Вивчення роботи паралельних портів Atmega328P 13

Лабораторна робота 2.
Вивчення зовнішніх переривань Atmega328P 27

Лабораторна робота 3.
Вивчення режимів роботи вбудованих таймерів-лічильників в Atmega328P 35

Лабораторна робота 4.
Використання вбудованого USART для взаємодії мікроконтролера та персонального комп’ютера 50

Лабораторна робота 5.
Аналогово-цифрове та цифро-аналогове перетворення 65

Лабораторна робота 6.
Використання вбудованого модуля SPI 80

Лабораторна робота 0.

Знайомство з програмними та апаратними засобами програмування мікроконтролерів AVR.

Мета: навчитися використовувати програмний пакет AVR Studio 7 для розробки та налагодження програм для AVR мікроконтролерів, на прикладі ATmega328P.

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 вбудованими (embedded) контролерами, оскільки МК часто вбудовані у пристрої та системи, які вони контролюють.

МК ATmega328P – це складний пристрій, в якому поєднано центральний процесор (ЦП), різні види пам'яті (енергонезалежні flash пам'ять для програм та EEPROM і SRAM для даних) і велика кількість периферійних пристроїв – таймери-лічильники, аналогово-цифровий перетворювач, компаратор, комунікаційні модулі (USART, SPI), тощо. Для того, щоб МК виконував корисну роботу, необхідно в його пам'ять записати послідовний набір команд – програму. Програму для МК створюють за допомогою персонального комп'ютера (ПК) використовуючи спеціальні програмні пакети.

В роботі буде використано інтегрована платформа розробки (ІПР) *Atmel Studio 7* яка дозволяє створювати, компілювати та налагоджувати програми для МК із використанням мови C/C++. В цей програмний пакет інтегровано редактор тексту програм, компілятор C/C++, симулятор роботи МК та програматор.

Atmel Studio 7 є доступним у вільному доступі (безкоштовно) на сайті виробника мікроконтролерів AVR компанії ***Microchip Technology Inc***. Для встановлення *Atmel Studio 7* потрібно виконати наступне: завантажити інсталяційні файли за адресою:

http://www.microchip.com/avr-support/atmel-studio-7,

запустити інсталяційні файли і, під час установки програмного пакету, вибрати потрібну архітектуру мікроконтролера "*Select Architecture*: AVR 8-bit MCU". Довідкову інформацію про програмний пакет *Atmel Studio 7* можна знайти на сайті виробника:

https://www.microchip.com/webdoc/

Використання Atmel Studio для програмування мікроконтролерів на мові С.

При навчанні програмування на ПК традиційно результатом роботи першої програми є виведення довільного тексту (як правило це "Hello Wold") на екран, при програмуванні МК традиційно першою програмою є програма управління світлодіодом. Створимо програму, яка буде включати світлодіоди відповідно до номеру натиснутої кнопки. Для цього потрібно виконати такі чотири кроки:

1. створити в *Atmel Studio* проект і написати програмний код на С, який є реалізацією алгоритму;
2. провести процес компіляції програми (компіляція – це перетворення програмного коду написаного на мові високого рівня С в сукупність команд які буде виконувати МК), отриманий файл, із розширенням *hex,* який буде завантажений в пам'ять МК для виконання;
3. виконати симуляцію роботи програми та провести налагодження (debugging) програми;
4. завантажити *hex*-файл в пам'ять МК ATmega328P і запустити її на виконання.

Крок 1. Створення робочого проекту в Atmel Studio 7.

- Для створення проекту запускаємо *Atmel Studio 7,* і на початковій сторінці вибираємо *New Project,* або використовуємо меню ***File→New Project*** (рис. 1).
- У діалоговому вікні яке з'явиться (рис. 2.) вибираємо "***GCC C Executable Project***" та вказуємо ім'я, та розташування проекту, відмічаємо "Create the directory for solution" буде створена окрема папка для проекту.

- Тиснемо кнопку “***OK***”.
- В діалозі “***Device selection***”, який з’явиться на екрані (рис. 3), знаходимо потрібний МК ATmega328P і тиснемо кнопку “***OK***”.

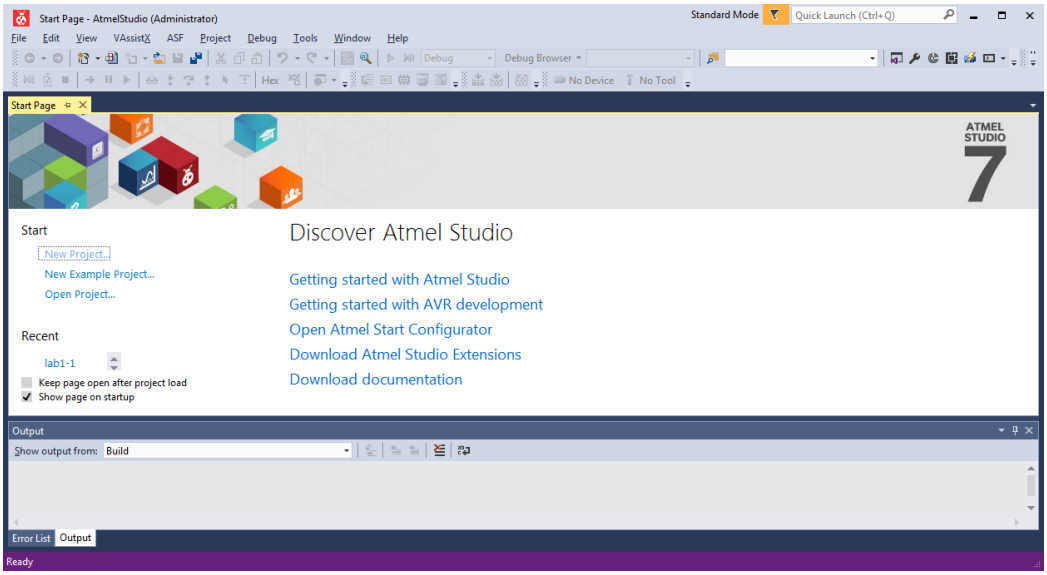


Рис. 1. Початкова сторінка *Atmel Studio 7.*

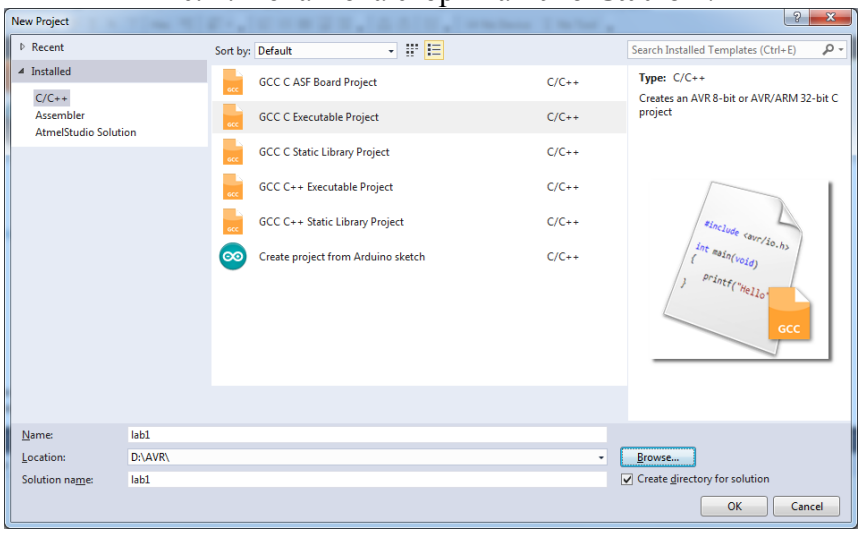


Рис. 2. Діалогове вікно вибору типу проекту, його імені та розташування.

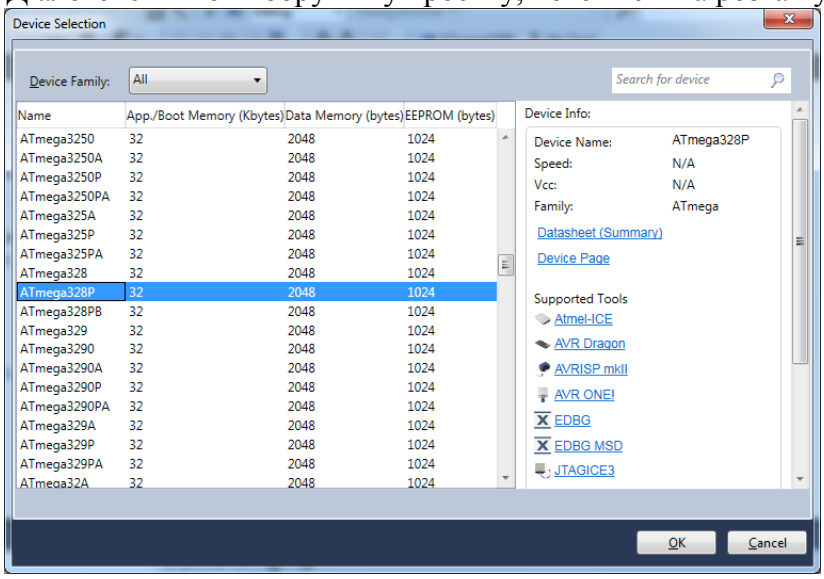


Рис.3 Діалогове вікно вибору типу мікроконтролера

Після цього буде створений проект ***lab1*** із початковим програмним кодом ***main.c*** (рис. 4.).

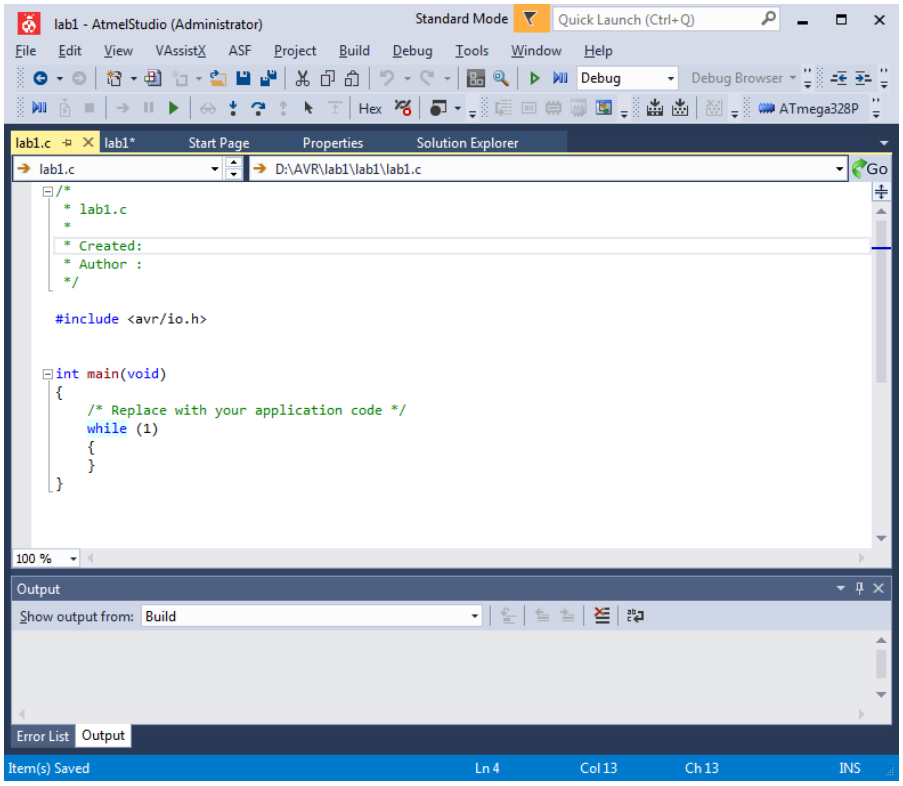


Рис. 4. *Atmel Studio 7* із відкритим проектом ***lab1.***

Крок 2. Створення С програми в Atmel Studio 7.

Тепер можна створювати програму для мікроконтролера. Нехай у нас така задача: на навчальному стенді (див. Додаток 1) є в наявності 6 кнопок та лінійка із 10-ти світлодіодів, при натисканні на кнопку світлодіоди відображають номер кнопки: якщо натиснута 1-ша кнопка, то світиться 1-й світлодіод, і так далі. Введіть програмний код (лістинг 1) в файл ***main.c***, можливо цей код не дуже зрозумілий, але це не так важливо в цій лабораторній роботі (роботу паралельних портів вводу-виводу, взаємодію із зовнішніми пристроями буде ґрунтовно розглянуто в лабораторній роботі №1 «Вивчення роботи паралельних портів Atmega328P»), а для кращого розуміння програмного коду зверніть увагу на коментарі.

Після введення коду збережіть всі файли проекту меню: ***File→Save All***, або використайте комбінацію клавіш: `Ctrl+Shift+S`

Лістинг 1. Відображення номеру кнопки

```
/*
 * lab1.c
 *
 * Приклад програми: відображення номеру кнопки за допомогою світлодіодів
 * при натисканні кнопки на світлодіодній лінійці світиться світлодіод із відповідним
номером
 */
#include <avr/io.h>  // бібліотечний заголовний файл де описані порти вводу-виводу МК AVR

int main(void)
{
// початкові налаштування портів вводу-виводу МК ATmega328P:
// кнопки підключені до PORTC, а світлодіоди підключенні до PORTD
// тому PORTC працює як вхід - перевіряє значення напруги на кнопках:
//  якщо напруга ~5 В (це відповідає логічній 1) то кнопка НЕ натиснута!
// відповідно якщо кнопка була натиснута то напруга ~0 В (логічний 0) - інверсна логіка.
// до PORTD під'єднані світлодіоди - вихід
// якщо в PORTD записати 1-ці то на виході буде напруга 5 В і світлодіоди будуть світитися
// змінна де буде зберігатися номер кнопки, volatile – вказує компілятору, що змінна
// може змінюватися не тільки в тексті програми
```

```
        volatile unsigned char i=0;
        DDRC=0x00;                  // налаштовуємо PORTC  як вхід
        PORTC=0xFF;                 // включаємо Pull-up резистори
        DDRD=0xFF;                  // налаштовуємо PORTD  як вихід

        while (1)  // нескінченний цикл де буде виконуватися основна задача
    {
        // зчитуємо стан PORTC і накладаємо маску - нам потрібні лише перші 6 розрядів
        i=PINC | 0b11000000;
                if (i!=0xFF)   // перевіряємо чи була натиснута кнопка
                {
                        PORTD=~i; // інвертуємо значення та записуємо в PORTD
                }
    }
}
```

Крок 3. Компіляція С програми та отримання hex-файла в Atmel Studio 7.

Для проведення компіляції виберіть меню **Build→Build Solution,** або «гаряча» кнопка **F7** (рис. 5).

Якщо відсутні повідомлення про помилки, то після завершення процесу компіляції у вкладці **“Output”** з’явиться повідомлення **“Build succeeded”** і буде створений файл із назвою lab1.hex**.** У вкладці **“Output”** відображається процес компіляції, а також там можна знайти інформацію про використання пам’яті МК:

Лістинг 2. Використання пам’яті МК (вкладка **“Output”**)

```
Program Memory Usage :      154 bytes   0,5 % Full
Data Memory Usage:          0 bytes   0,0 % Full
```

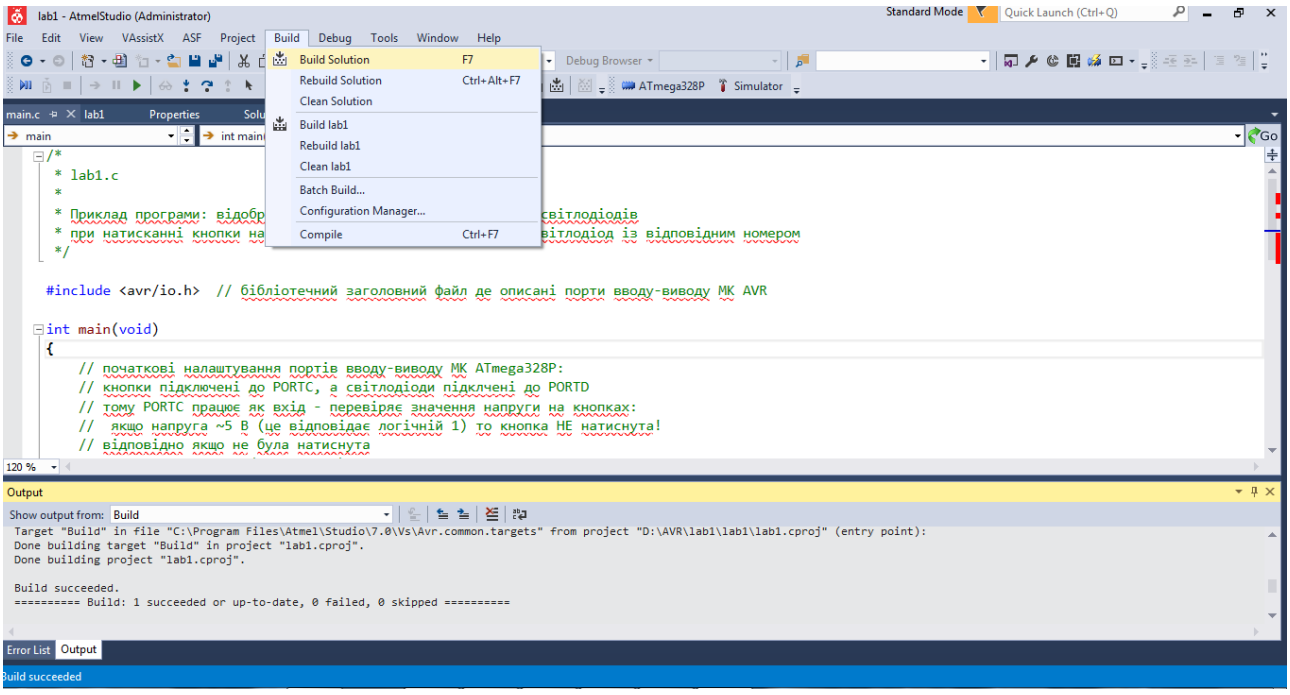


Рис. 5. Компіляція та вкладка “**Output**”

У випадку, коли в програмі присутні синтаксичні помилки (внесемо помилку – приберемо знак «;» в кінці рядка) , то після компіляції буде активна вкладка “**Error List**” (рис. 6), де буде перелік та тип помилок і вказано номер рядка, де розташована помилка в програмі. Якщо два рази клацнути лівою кнопкою миші, то курсор буде переведений на те місце в програмі, де знаходиться помилка.

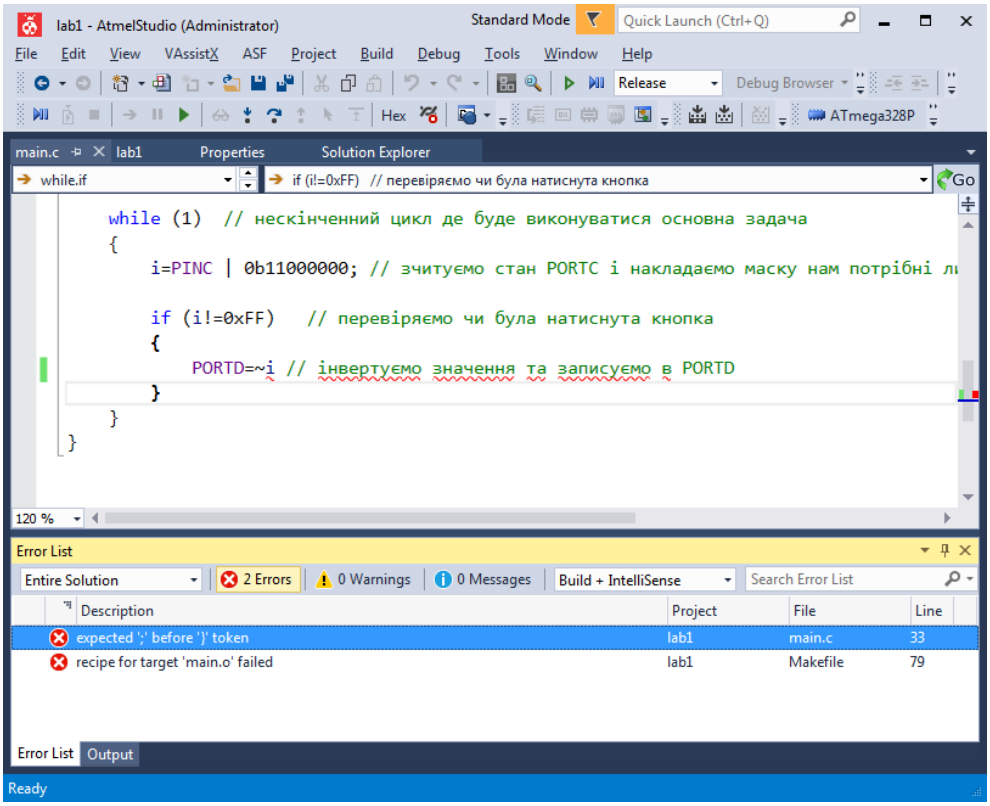


Рис. 6. Наявні помилки відображаються у вкладці "**Error List**"

Після виправлення синтаксичних помилок можна переходити до наступного етапу – налагодження програми та виявлення логічних помилок.

Крок 4. Налагодження (debugging) С програми за допомогою симулятора в Atmel Studio 7.

Процес налагодження програми є важливим етапом в програмуванні – за його допомогою виявляються логічні помилки в роботі програми. В цьому кроці розберемо, як проводити налагодження програми на рівні С коду використовуючи засоби *Atmel Studio*. Основний принцип – виконання одного рядка програми і після цього спостерігаємо, як змінюється вміст комірок пам'яті, регістрів центрального процесора, портів вводу-виводу. Це виконується за допомогою спеціальної програми - симулятора роботи МК AVR. Проведемо симуляцію роботи проекту ***lab1***, для цього виконаємо такі кроки:

- переключимо на вкладку із назвою проеку `lab1`, або виберемо меню **Project→lab1 Properties** (комбінація клавіш **ALT+F7**) і в меню **Tools→Selected debugger/programmer** вибираємо **Simulator** (рис. 7);
- запускаємо симуляцію роботи МК ATmega328P за допомогою меню **Debug→Start Debugging and Break** (Alt+F5);
- програмний симулятор *Atmel Studio* дозволяє перевіряти стан та вміст регістрів процесорного ядра (CPU), вміст регістрів периферійних пристроїв та портів вводу-виводу. Щоб включити цю опцію потрібно вибрати в меню **Debug→Windows,** а потім вибрати **Processor View** або **I/O View** тоді з'являться вікна **Processor Status,** або **I/O** (рис. 8);
- покрокове виконання програми виконується за допомогою команди **Step Into** в меню **Debug→Step Into (F11),** та рядок С коду, який буде виконуватися наступним, виділяється жовтим кольором і на нього вказує жовта стрілка (рис. 9);
- у вікні **I/O** можна спостерігати результат виконання команди, наприклад: після виконання команди

```
DDRD=0xFF;              // налаштовуємо PORTD  як вихід
```

в регістрі DDRD всі біти будуть встановлені в 1 – це відповідає числу 0xFF.

Під час симуляції роботи програми вміст комірок пам’яті та регістрів може бути змінений вручну, наприклад: оскільки до PORTC підключені кнопки, то для імітації їх роботи стан регістрів потрібно змінити в ручному режимі. Для цього виберіть мишкою PORTC у вікні **I/O** і, щоб змінити значення бітів в регістрі PINC**,** просто клацніть мишкою на відповідний біт (біти позначені у вигляді квадратів: пустий квадрат – 0, а зафарбований – 1, рис. 10, а). Внесені зміни будуть враховані при подальшому виконанні програми.

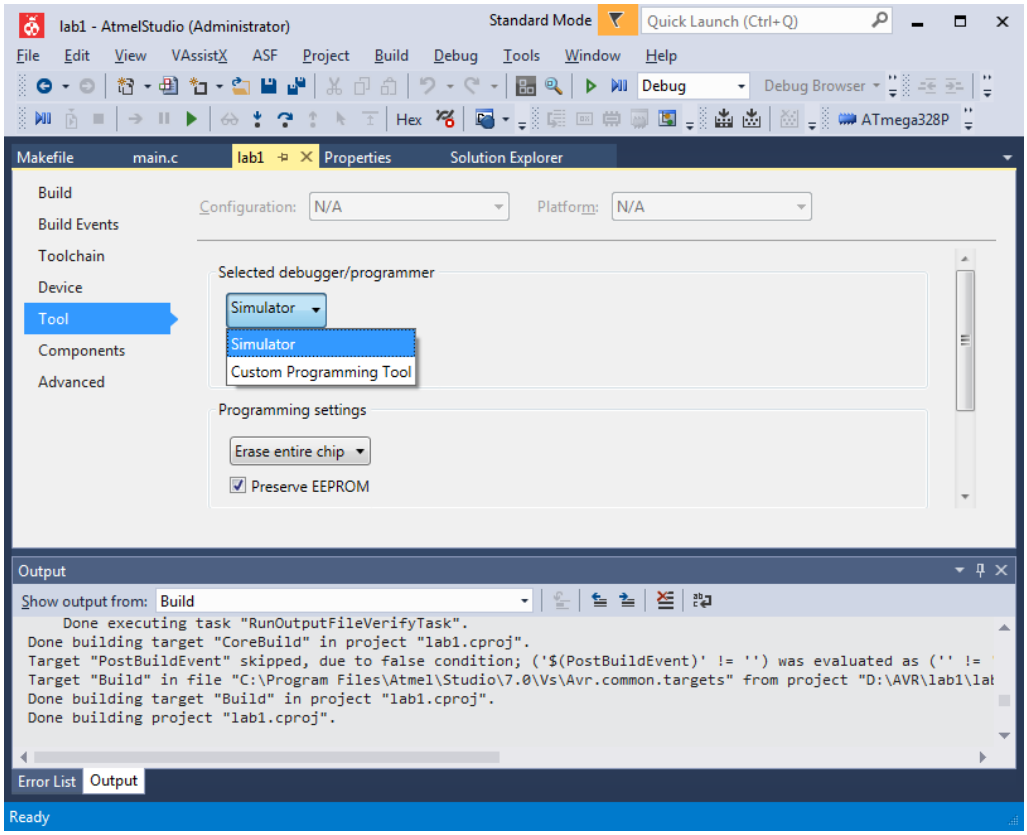
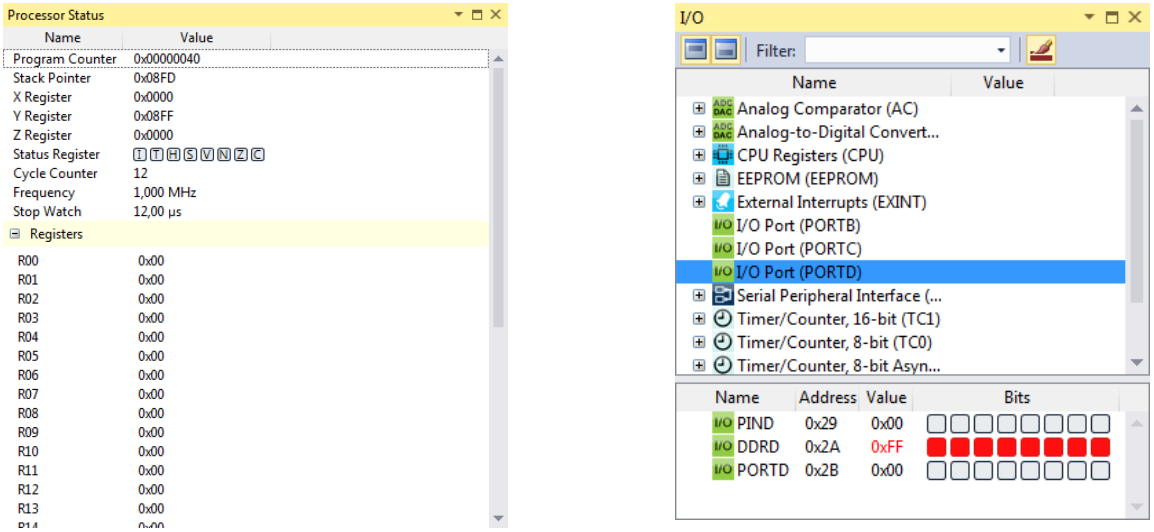


Рис. 7. Встановлення програмного симулятора для налагодження програми.



а б

Рис. 8. Вікна **Processor Status** (а) та **I/O** (б)

Для спостереження значень змінних в програмі використовуються вікна **Watch,** для того, щоб додати змінну до вікна **Watch**, потрібно вибрати в меню **Debug→QuickWatch**

(гарячі клавіші **Shift+F9)** і додати потрібну змінну. Під час виконання програми поточне значення змінної буде відображатися (рис. 10, б)

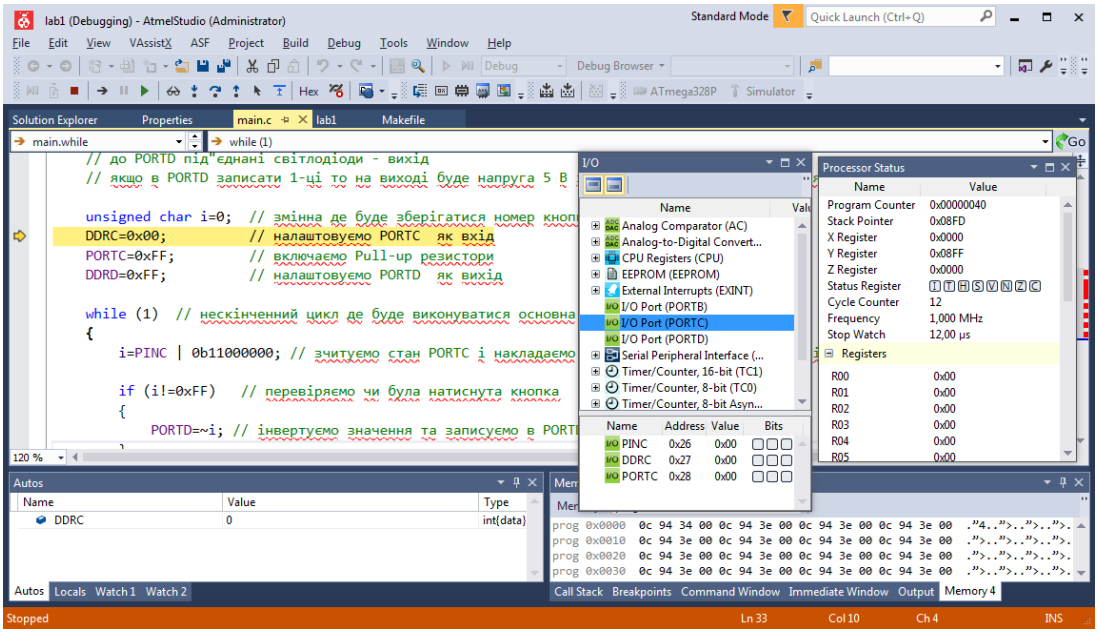
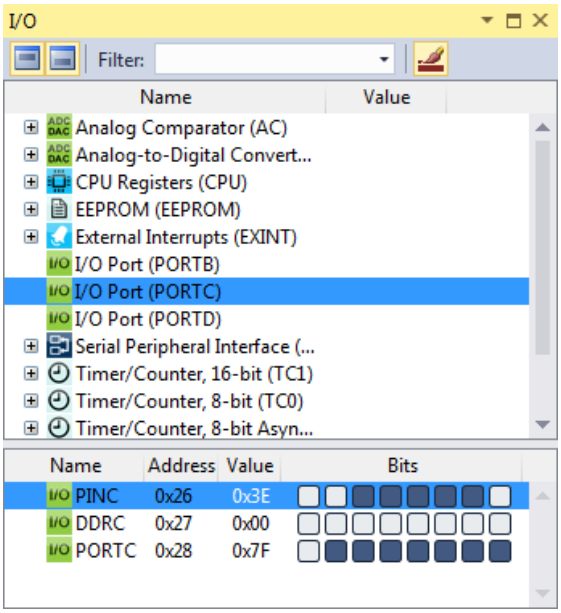
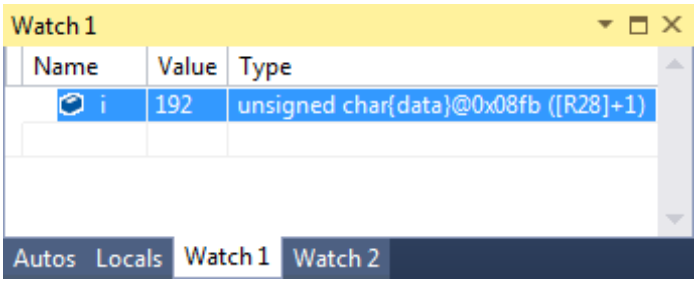


Рис. 9. Покрокове виконання С коду програми.



а



б

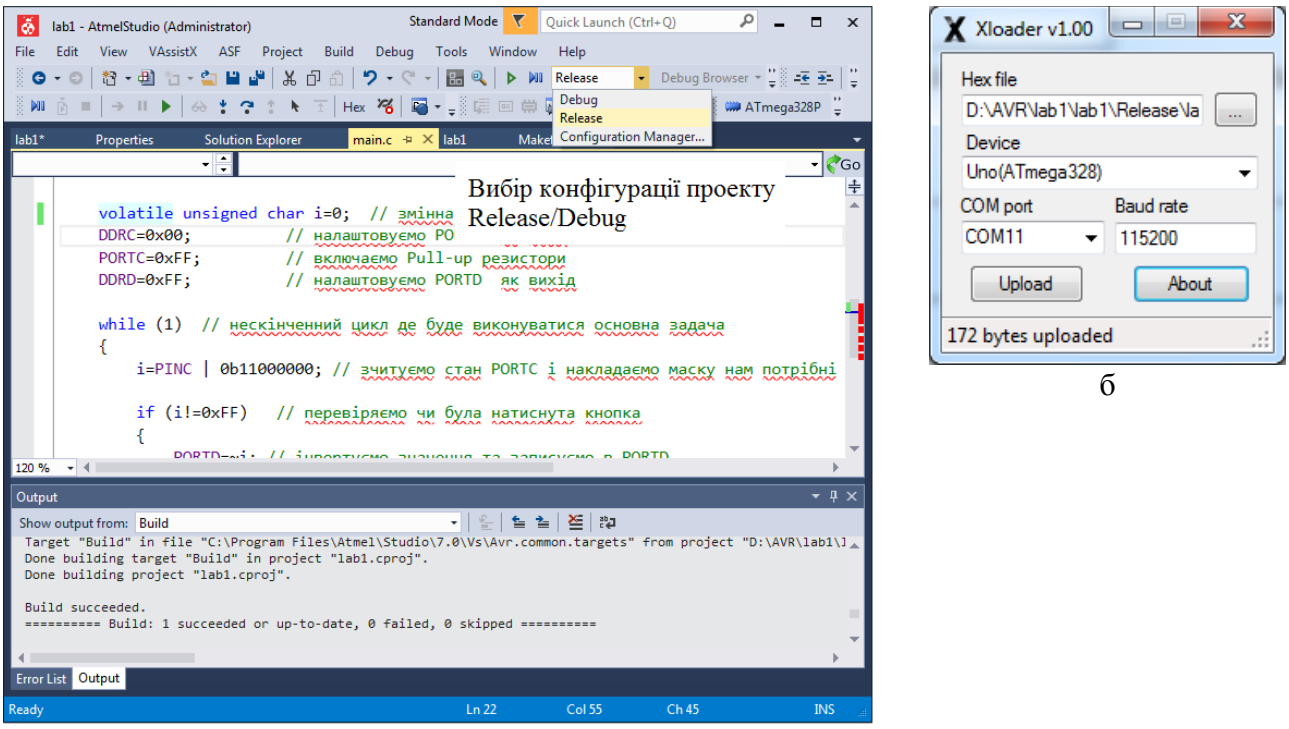
Рис. 10. Зміна регістрів в ручному режимі (а), та вікно **Watch1** для спостереження значення змінних.

Меню **Debug** забезпечує багато різних можливостей для симуляції роботи програми: встановлення точок зупинки (Break Point), покрокове виконання програми, коли функція розглядається, як один оператор – команда **Step Over, (F11**), виконання програми до поточного положення курсору – **Run to Cursor (Ctrl+ F10)**, тощо. Для порівняння Асемблерного коду паралельно до С коду можна використати меню **Debug→Windows→Disasembly.**

Для зупинки процесу симуляції програми вибираємо меню **Debug→Stop Debugging.**

Крок 5. Завантаження hex-файла в пам'ять мікроконтролера ATmega328P.

Для отримання програми в машинних кодах, яку буде виконувати МК потрібно провести компіляцію проекту (**Build→Build Solution,** або **F7,** рис. 5), але попередньо вибрати конфігурацію проекту **Release** (рис. 11.)



а б

Рис. 11. Вибір конфігурації проекту **Release/Debug** (а), завантаження програми в МК ATmega328 за допомогою Xloader (б)

Після цього в папці проекту з'явиться папка із назвою **Release,** де будуть розміщені файли отримані в результаті компіляції, поміж них буде файл *lab1.hex* в якому зберігається програма в машинних кодах, яка записана в шістнадцяти розрядній системі числення (лістинг 3).

Лістинг 3. Файл *lab1.hex*

```
:100000000C9434000C943E000C943E000C943E0082
:100010000C943E000C943E000C943E000C943E0068
:100020000C943E000C943E000C943E000C943E0058
:100030000C943E000C943E000C943E000C943E0048
:100040000C943E000C943E000C943E000C943E0038
:100050000C943E000C943E000C943E000C943E0028
:100060000C943E000C943E0011241FBECFEFD8E04C
:10007000DEBFCDBF0E9440000C9454000C940000E1
:10008000CF93DF931F92CDB7DEB7198217B88FEFEA
:1000900088B98AB986B1806C898389818F3FD1F311
:0C00A000898180958BB9F6CFF894FFCFD2
:00000001FF
```

Для того, щоб завантажити файл *lab1.hex* в МК, використаємо програму **AVR Dude** із графічною оболонкою **Xloader**. В якості плати контролера використовується плата Arduino Uno із встановленим МК ATmega328P та перетворювачем USB-USATR FT232, або CH240G для забезпечення зв'язку між МК та персональним комп'ютером. Для завантаження *lab1.hex* файла в МК запускаємо **Xloader**, в меню **Hex file** вказуємо шлях до файла (рис. 11, б), вибираємо пристрій **Device→Uno(ATmega328)** і тиснемо кнопку **Upload.** Після успішного завантаження з'явиться повідомлення "172 bytes uploaded", якщо з'явиться повідомлення "Can't open port", то потрібно в меню **COM port** вибрати активний COM port.

Для перевірки роботи програми, потрібно на лабораторному стенді під’єднати до PORTC 6 наявних кнопок, а до PORTD світлодіоди. Натискаючи на кнопки перевіряємо роботу програми (рис. 12).

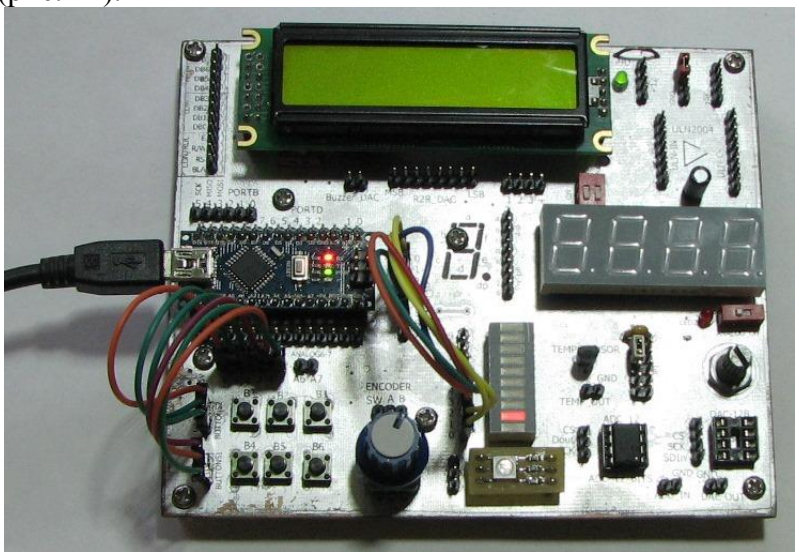


Рис. 12. Лабораторний стенд.

Завдання до лабораторної роботи:

1. Встановіть на персональний комп’ютер *Atmel Studio 7.*
2. Самостійно створіть новий проект в *Atmel Studio 7.*
3. Використовуючи вбудований редактор надрукуйте текст програми, наведений на листингу 1.
4. Виконайте компіляцію проекту в конфігурації Debug та проведіть симуляцію роботи програми.
5. Виконайте компіляцію в конфігурації Release.
6. На лабораторному стенді під’єднайте кнопки та світлодіоди до відповідних портів .
7. Використовуючи програму **Xloader,** завантажте hex-файл в ATmega328P та перевірте роботу програми.

Лабораторна робота 1.

Вивчення роботи паралельних портів Atmega328P

Мета: Вивчення структури та характеристик паралельних портів вводу-виводу (ПВВ) загального МК ATmega328P. Вивчення методів налаштування та програмування ППВ.

Кожен МК має засоби для взаємодії із зовнішнім оточенням – генерувати керуючі електричні сигнали та сприймати і обробляти зовнішні сигнали. Якщо ці сигнали є цифровими (цифрові електричні сигнали мають два чітко визначені діапазони напруг – один діапазон відповідає логічній «1», а інший логічному «0»), то використовують цифрові лінії вводу-виводу. Цифрові лінії можуть створювати електричні сигнали, які відповідають логічному «0» чи «1», або перевіряти напругу, яка прикладена до лінії на відповідність логічному «0» чи «1». Для зручності роботи цифрові лінії об'єднують – таку сукупність цифрових ліній називають паралельний порт вводу-виводу (In/Out Port). Оскільки МК AVR є 8-ми бітними, то кількість ліній в ПВВ не перевищує 8 (може бути менше). В МК AVR реалізований спосіб функціонування ПВВ «читання/модифікація/запис» окремо для кожної лінії, тобто зміна напряму роботи (лінія є входом чи виходом) відбувається незалежно від стану інших ліній порту. Порти прийнято позначати літерами A, B, C, D, …

МК є складний пристрій, який виконує задану програму, та може взаємодіяти із зовнішніми пристроями, тому для використання ПВВ, потрібно знати їх електричні характеристики та програмні методи роботи із ними.

Основне джерело інформації про характеристики та структуру МК є технічна документація виробника (Datasheet) на ATmega328P, вона доступна на сторінці виробника:

http://www.microchip.com/wwwproducts/en/ATmega328P .

Де наведений короткий опис МК ATmega328P, а в розділі *Documentation* є посилання на технічну документацію та численні документи, де описані приклади використання МК AVR. Ці документи називаються *Application Notes,* наприклад *AVR134: Real Time Clock (RTC) Using the Asynchronous Timer* – описаний спосіб реалізації годинника реального часу. Технічний опис МК можна завантажити за адресою:

http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf .

Цей документ є дуже важливий та інформативний, в ньому наявні не тільки опис всіх пристроїв, які є складовими МК, а також наведені приклади програмних кодів для їх використання. Детальний опис ПВВ наведений в розділі *18 I/O Ports* ст. 97. В ATmega328P є 23 лінії вводу-виводу, які згруповані в 3 ПВВ, які позначаються PORTB, PORTD, PORTC, лінії кожного порту під'єднані до виводів (лапок, pin) МК. Фізичне розташування лапок МК (рис. 1) та їх відповідність до ПВВ описано в розділі *5 Конфігурація виводів (Pin Configuration)* ст. 14., рис. 5-1 – 5-4, а в розділі 5-2. *Опис виводів* (*Pin Description*) наведено опис функцій кожного виводу.

Електричні характеристики портів вводу-виводу.

Електричні характеристики наведені в розділі: 32 *Електричні характеристики,* ст. 365. В таблиці *32-1 Абсолютні граничні значення*, наведені граничні значення сигналів для МК AVR. Перевищення значень наведених в цій таблиці може привести до незворотного пошкодження мікроконтролера. Безпечна напруга, яку можна прикладати до лапки МК повинна лежати в межах -0,5 В – V_{CC}+0,5 В, де V_{CC} – напруга живлення. Максимально можлива напруга живлення 6 В, а робочий діапазон напруги живлення 1,8 В – 5,5 В. Максимально допустимий струм, який може протікати через один вихід 40 мА, а загальний струм, який може протікати через лапки МК (через лінії живлення чи землі) не повинен перевищувати 200 мА. В наступній таблиці 32-2 *Загальні характеристики по постійному струму (Common DC Characteristics)* описані діапазони напруг, які будуть сприйматися як логічний «0» (низький рівень) та логічна «1» (високий) рівень:

-0,5 В – 0,3V_{CC} – низький рівень;

$0{,}7V_{CC}$ – V_{CC}+0,5 В – високий рівень.

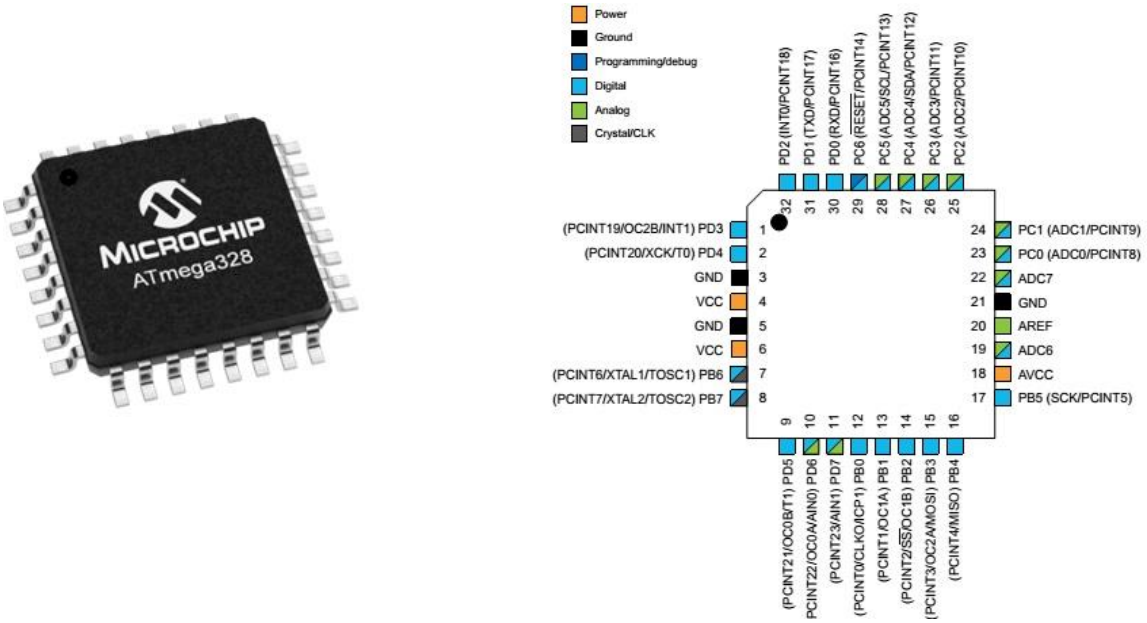


Рис. 1. МК ATmega328 в корпусі TQFP (а), відповідність виводів МК до їх функціонального призначення.

Для уникнення багаторазових переключень логічного стану на границі діапазону високий-низький рівень, присутній гістерезис переключення. Перехід від низького рівня до високого відбувається за однієї напруги, а зворотній перехід високий-низький при дещо меншій напрузі. Розмах гістерезисної петлі складає 0,05 V_{CC}.

Дещо спрощена структурна схема лінії ПВВ наведена на рис.2.

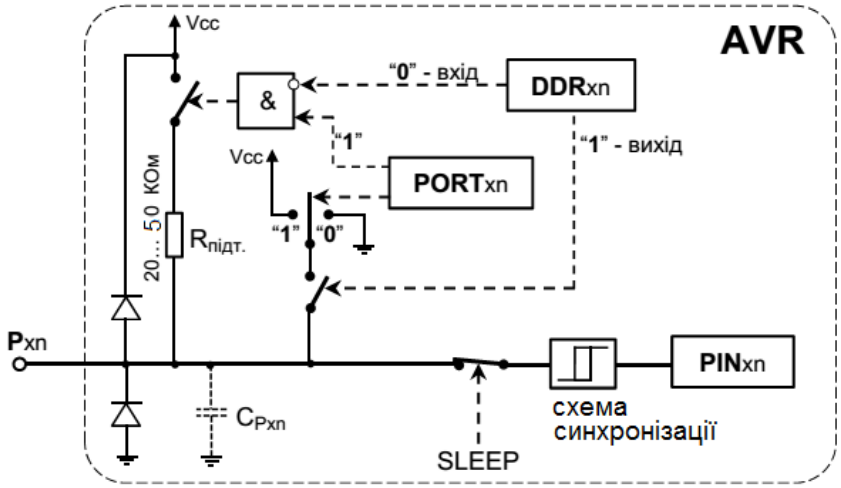


Рис. 2. Структурна схема лінії вводу-виводу портів МК AVR.

Усі лапки портів мають внутрішній діодний захист, який захищає від стрибків підвищеної напруги (верхній діод на схемі буде відкриватися при напрузі вищій за напругу живлення V_{CC}) та від'ємної напруги, яка нейтралізується на землю нижнім діодом. Цей діодний захист призначений для знешкодження імпульсних завад, і при постійній напрузі, яка перевищує граничні допустимі значення лінія порту вийде з ладу. На кожній лапці присутня невелика паразитна ємність (позначене пунктиром) величина якої не перевищує 10 пФ. Лінії входу-виходу є симетричними вони можуть бути, як джерелом струму, так і споживачем. На всіх лініях присутні «підтягуючі» (pull-up) резистори (величина їх знаходиться в межах від 20 до 50 кОм), резистори можуть бути підключені окремо для кожної лінії.

Мікроконтролери, і МК AVR не є виняток, проектують із прицілом на максимальну гнучкість так, щоб за допомогою одного МК можна було виконувати широкий спектр задач. До складу МК включають велику кількість периферійних пристроїв (порти вводу-виводу, таймери-лічильники, аналогово-цифровий перетворювач тощо), які перед використанням

потрібно налаштувати на потрібний режим роботи. Налаштування проводиться програмним шляхом. З програмної точки зору кожен периферійний пристрій (ПП) являє собою сукупність 8 чи 16-ти розрядних регістрів (комірок пам'яті), які можна розділити на дві категорії: контрольні/управляючі та регістри даних. Контрольні/управляючі регістри призначені для налаштування периферійного пристрою на потрібний режим роботи, а в регістрах даних зберігаються результати роботи цього пристрою. Сукупність біт, які записані в контрольні регістри, визначають вид роботи периферійного пристрою, тому процес налаштування (початкової ініціалізації) зводиться до запису відповідних бітів в контрольні/управляючі регістри. Управляючі регістри та конфігурації управляючих бітів для кожного периферійного пристрою детально описані в технічному описі ATmega328P. Оскільки режими роботи ПП задаються програмно, то це дає змогу змінювати налаштування, режими роботи ПП під час виконання програми МК.

За керування лініями ПВВ відповідає сукупність 3-х регістрів які розташовані в пам'яті МК. Налаштування та робота із ПВВ зводиться до обміну даними із цими регістрами. Детальний опис регістрів ПВВ наведений в технічному описі розділ 18.4. *Register Description* ст.114.

Регістри ПВВ:

DDRx – регістр, який визначає напрямок роботи ПВВ;

PORTx – регістр даних;

PINx – регістр вхідних даних.

Назви регістрів наведені в загальному вигляді: x відповідає назві ПВВ – A, B, C, D, …, а для позначення конкретної лінії порту будемо використовувати позначення Pxn, де x назва порта, а n – номер лінії може бути від 0 до 7.

DDRx - це регістри, що визначають напрямок роботи ліній порту: на вхід чи на вихід. При цьому можна налаштовувати індивідуально кожну лінію вибраного порту. Цей регістр є двонаправлений. Дані в нього можуть бути як записані, так і прочитані. Якщо біт в регістрі **DDRx,** який відповідає потрібній лапці порту рівний «0», то вона працює на вхід, якщо «1» то на вихід.

PORTx – регістр даних порту, який являє собою вихідний буфер: якщо лінія порту налаштована на вихід (відповідний біт в регістрі **DDRx** встановлений в «1»), то після запису «1» в відповідний біт в **PORTx** на лапці МК з'явиться рівень напруги, який відповідає високому рівню близький до V_{CC} (напруги живлення). Значення «0» встановлює низький рівень напруги – близький до GND (0 В цифрова земля). Якщо лінія порту працює на вхід (відповідний біт в **DDRx** встановлений в «0»), то **PORTx** буде визначати її електричні характеристики. Значення «1» у відповідному біті **PORTx** підключить підтягуючий резистор $R_{підт}$ (рис. 1), номінал резистора 20 – 50 кОм, точне значення невідоме, і на лапці буде присутня V_{CC}. Якщо записано «0», то вивід МК буде перебувати в стані Hi-Z: повний опір буде дуже великий і не впливатиме на роботу зовнішніх пристроїв.

Розглянемо порт B, нехай в регістрі **DDRB** та **PORTB** записані біти (V_{CC}=5 В):

DDRB:

n, номер біта	7	6	5	4	3	2	1	0
значення	1	1	0	0	0	1	1	0
позначення та вид роботи	PB7 Out	PB6 Out	PB5 In	PB4 In	PB3 In	PB2 Out	PB1 Out	PB0 In

PORTB:

n, номер біта	7	6	5	4	3	2	1	0
значення	1	0	1	1	0	0	1	0
напруга та вид роботи	5 В Out	0 В Out	5 В In	5 В In	Hi-Z In	0 В Out	5 В Out	Hi-Z In

Ті лінії порту, які налаштовані на вихід на них буде висока 5 В (PB7, PB1) чи низька 0 В (PB6, PB1) напруга і вони можуть бути джерелом чи споживачем струму. На лініях PB5, PB4 теж буде присутній високий рівень напруги, але вихідний струм буде обмежений резисторами підтяжки $R_{підт}$.

PINx. Незалежно від налаштування регістра **DDRx**, реальний стан лінії порту (рівні напруги прикладені до відповідної лапки МК) може бути прочитаний через біти регістра **PINxn.** Для стабільності зчитування стану виводів МК присутня схема синхронізації, яка складається із тригера та розряду регістра **PINxn** (рис. 2.). Значення зовнішньої напруги фіксується в трігері за низького рівня тактового сигналу МК і перезаписується в **PINxn** за переднього фронту тактового сигналу. Тому, між операціями запису у порт та читанням із нього потрібно вводити затримку.

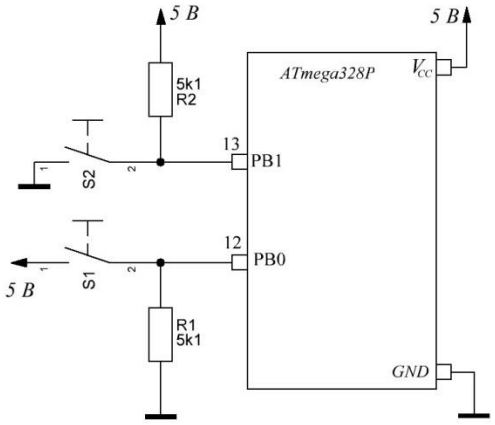
Значення бітів в регістрі **PINx** відповідають реальним рівням сигналів на виводах МК: «1» - при високому рівню напруги, а «0» - при низькому. Якщо підключені підтягуючі резистори, то в **PINx** будуть записані «1». Щоб отримати «0» потрібно лапку МК під'єднати до сигналу GND. Якщо лінії порту знаходяться в Hi-Z стані і відсутні зовнішні сигнали, то значення **PINxn** будуть «0». Однак, за присутності найменших електричних завад, наприклад дотик пальця до лапки МК, на виході з'явиться «1». Тому, для уникнення впливу завад, потенціали ліній порту, які працюють на вхід, повинен бути визначений – лінію через резистори під'єднують до V_{CC} або до GND.

Регістр **PINx** має ще одне цікаве застосування, якщо ПВВ налаштований на вихід, то записавши біт «1» в **PINxn** значення на лінії зміниться на протилежний. Таким чином можна формувати змінний, періодичний сигнал використовуючи лише одну команду.

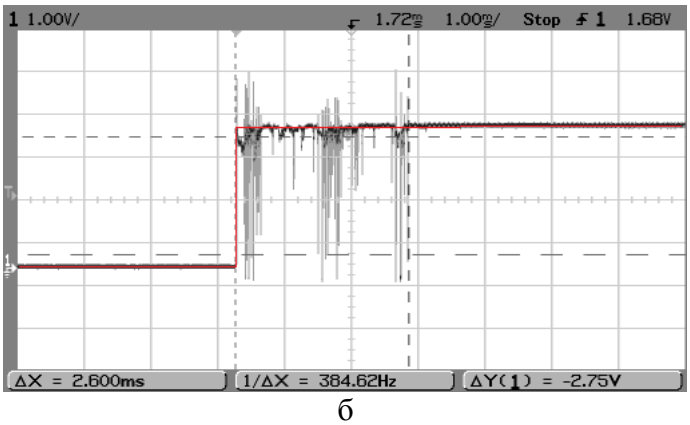
Використання паралельних портів вводу-виводу.

Підключення на використання механічних кнопок та перемикачів.

Як правило, мікроконтролери є вбудованими в електронні системи і події фіксуються за допомогою різноманітних давачів. Давачі можуть бути аналогові (наприклад термоопір: при зміні температури його опір змінюється неперервно), або цифрові (вимикачі, кнопки тощо). Цифрові давачі мають тільки два визначених стани, які умовно можна позначити «Так/Включено», «Ні/Виключено» та можна приписати логічні значення «1» чи «0». Для контролю стану таких давачів можна використовувати лінії ПВВ. Паралельні порти дозволяють створювати управляючі сигнали та генерувати статичні та імпульсні цифрові сигнали. Розглянемо принцип взаємодії із типовим двійковим давачем – механічною кнопкою. Часто кнопки та перемикачі становлять основу зовнішнього інтерфейсу мікроконтролерної системи. Механічна кнопка являє собою пару електричних контактів, які можуть замикатися чи розмикатися при прикладені зусиль до клавіші кнопки. Існує багато варіантів підключення кнопки до МК. Розглянемо декілька найпоширеніших. Для генерації цифрового електричного сигналу кнопку потрібно під'єднати до ліні живлення V_{CC} та до землі GND. Типові схеми підключення наведено на рис. 3, а.



а



б

Рис. 3 а) Типове підключення кнопок до паралельного порту МК; б) «брязкіт» контактів механічної кнопки.

Кнопка S1 під'єднана до напруги живлення та до лапки МК №12, яка відповідає лінії PORTB PB0, лінія PB0 через резистор R1 підтягнута до землі. Якщо кнопка не натиснута, то на PB0 присутній низький рівень напруги. При натисканні кнопки S1 через неї та резистор R1 буде

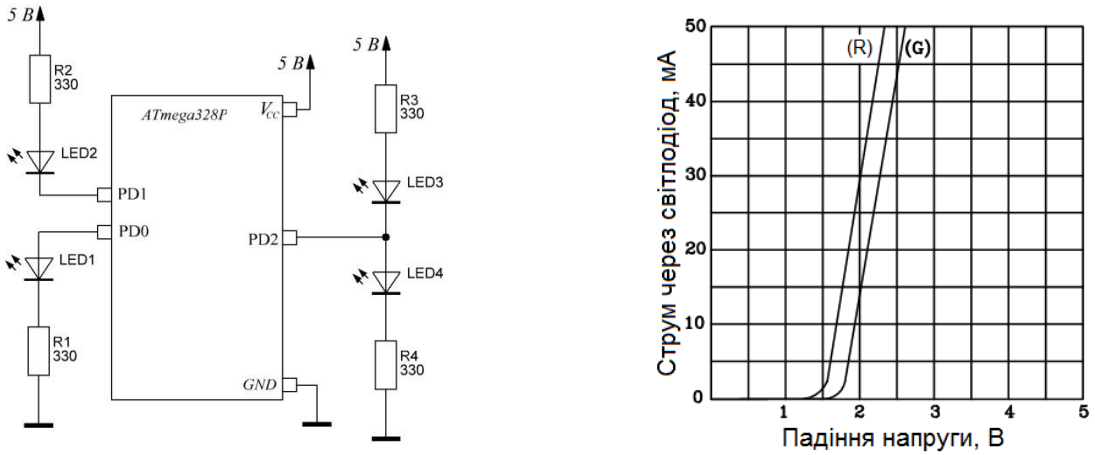
протікати струм і напруга на PB0 буде рівна падінню напруги на R1, тобто V_{CC}. Натиснутій кнопці буде відповідати логічна «1», а не натиснутій – «0». Така логіка роботи пристрою називається пряма логіка. Кнопка S2 під'єднана до GND та до лапки МК №13, яка відповідає лінії PORTB PB1, лінія PB1 через резистор R2 підтягнута V_{CC}. Якщо кнопка S2 не натиснута, то на PB1 присутній високий рівень напруги – логічна «1», а при натисканні кнопки S2 відбувається перехід від високого рівня до низького рівня напруги («1» → «0») – інверсна логіка. Номінали резисторів R1, R2 вибираються із розрахунку, щоб через кнопки протікав струм ~1 мА.

При переключенні механічних кнопок чи перемикачів виникає явище, яке полягає в небажаному і випадковому багатократному замиканні та розмиканні електричних контактів в момент комутації. Це явище називають «брязкіт» контактів (*contact bounce*). Внаслідок брязкоту контактів в електричному колі виникають імпульсні завади, які тривають від одиниць до сотні мілісекунд – це залежить від конструктивних особливостей контактної групи кнопки. Осцилограма брязкоту контактів наведена на рис. 2,б, де червоним кольором позначено переключення «ідеальної» кнопки. Із осцилограми видно велику кількість переключень протягом 2,6 мс. Оскільки тактова частота роботи МК є великою - 16 МГц, то МК буде сприймати завади, як багаторазове натискання кнопки. Для усунення брязкоту контактів можна застосовувати як схемотехнічні (фільтрація за допомогою *RC*-ланки, застосування тригерів Шмітта), так і програмні методи. Схемотехнічні методи вимагають ускладнення електричної схеми та додаткових електронних компонент, тому ми їх розглядати не будемо. Розглянемо програмний метод, алгоритм боротьби із брязкотом контактів:

- *фіксуємо натискання кнопки – зміна логічного стану лінії вводу-виводу порту;*
- *чекаємо 10-50 мс;*
- *якщо логічний стан лінії за цей час не змінився – кнопка натиснута, в протилежному випадку вважаємо, що це завади і кнопка не натиснута.*

Підключення світлодіодної індикації до МК.

Електричні характеристики портів МК AVR дозволяють підключачи світлодіоди безпосередньо до виводів мікроконтролера. Типові схема підключення світлодіодів наведені на рис. 4,а.



а б

Рис. 4. а) Підключення світлодіодів до МК, б) вольт-амперні характеристики світлодіодів червоного (R) та зеленого (G) кольору.

При підключенні світлодіодів потрібно дотримуватися полярності: анод підключається до високого потенціалу, а катод до низького потенціалу. Якщо різниця потенціалів анод-катод перевищує порогове значення, яке залежить від діоду, то буде протікати струм. Відповідно до цього, ми можемо по різному підключати світлодіоди: анод до лінії вводу-виводу МК, а катод до низького потенціалу GND (світлодіод LED1, рис. 4,а) або анод до високого потенціалу V_{CC}, а катод до лінії вводу-виводу (світлодіод LED2, рис. 4,а). В першому випадку світлодіод буде світитися, якщо на ліній високий потенціал

(записана «1» у відповідний розряд Pxn), а у другому на лінії повинен бути низький потенціал. Обидва способи є рівнозначними і визначається зручністю застосування. У деяких випадках можливе застосування 2-х світлодіодів різного кольору приєднаних до однієї лінії порту (LED3, LED4, рис. 4,а). Якщо подати «0» на вихід, то буде світитися LED3, а за «1» - світитиметься LED4. Якщо перевести вихід порту в Hi-Z стан, то струм буде протікати через обидва світлодіоди з'єднані послідовно, струм буде протікати значно менший, тому світлодіоди будуть світитися тьмяно. Цей метод дозволяє за допомогою однієї лінії вводу-виводу відображати 3 стани.

Опір резисторів біля світлодіодів буде визначати струм, який протікатиме через світлодіод і вивід МК. Очевидно, що струм не повинен перевищувати граничні допустимі значення як лінії МК, так і світлодіоду. Робочі та граничні струми світлодіоду описані в його технічній документації. Наприклад для діоду типу BL-BEG274 вказано граничний струм 30 мА, номінальний робочий 20 мА, пряме падіння наруги для світлодіоду зеленого свічення V_F=2,2 – 2,6 В, червоного V_F=2,0 – 2,6 В. Якщо виберемо струм діодів ~10 мА, падіння напруги за такого стуму знаходимо із вольт-амперної характеристики (рис. 4, б). Для червоного це буде V_F=~1,7 В, а для зеленого V_F=~1,9 В. Опір резисторів буде визначатися із рівняння Кірхгофа:

$$V_{CC} = IR + V_F \rightarrow R = \left(V_{CC} - V_F\right)/I$$

$$R = \left(5 - 1{,}9\right)/0{,}01 = 310\ Ом,$$

вибираємо найближчий номінал 330 Ом.

Створимо програму, яка буде із певним інтервалом включати та виключати світлодіод, а при натисканні на кнопку буде зменшувати період в два рази. при досягненні частоти блимання світлодіоду, яка вже не буде розрізнятися оком – встановити максимальний період в 1 с. При включенні живлення світлодіод світиться і для включення блимання потрібно натиснути кнопку. Програмний код такої програми наведено в лістингу 1.1

Лістинг 1.1. Приклад програми управління свіченям світлододу

```
#define F_CPU 16000000UL //тактова частота роботи МК

#include <avr/io.h>
#include <util/delay.h>

// Кнопку підключаємо до PORTC
#define Knopka_PORT PORTC
#define Knopka_DDR DDRC
#define Knopka_PIN PINC
#define Knopka PC2   // кнопку підключаємо до PC2 - вхід

// Світлодіод підключаємо до PORTD
#define LED_PORT PORTD
#define LED_DDR DDRD
#define LED_PIN PIND
#define LED_1 PD5  // світлодіод - PD5 – вихід

#define MAX_TIME 1000 // максимальний період 2*MAX_TIME в мс

//---------------- Прототипи функцій ----------------------------------------------------
void Init_Port(void); // Функція початкового налаштування ліній вводу виводу портів

// функція перевірки стану кнопки: кнопка натиснута, то повертається значення 1,не натиснута - 0
unsigned char Check_Button(unsigned char Button);
//---------------------------------------------------------------------------------------
//--------- Опис функцій -------------------------------------
void Init_Port(void)
{
	Knopka_DDR &=~(1<<Knopka);  // кнопку підключаємо до PC2 - вхід
	LED_DDR |=(1<<LED_1);       // світлодіод - PD5 - вихід
}
```

```
unsigned char Check_Button(unsigned char Button)
{
       // первіряємо чи була натиснута кнопка
       unsigned char foo =( (~Knopka_PIN) & (1<<Button) )>>Button;
       if (foo)
       {
              _delay_ms(20); // фільтруємо "брязкіт" контактів кнопки
              foo =( (~Knopka_PIN) & (1<<Button) )>>Button; // читаємо ще раз
       }
       return foo;
  }
int main(void)
{
       Init_Port();
       unsigned int delay_time=MAX_TIME;
       volatile char k=0;

       LED_PORT|= (1<<LED_1); // включаємо світлодіод

       while(!k) // чекаємо допоки буде натиснута кнопка
       {
              k=Check_Button(Knopka);
       }
       LED_PORT &= ~(1<<LED_1); // виключаємо світлодіод

    while (1) // початок виконання основної частини програми в нескінченному циклі
    {
 // --------------- створюємо періодичний сигнал ---------------------------------------
       unsigned int bar = delay_time;
       LED_PIN |= (1<<LED_1); // записуємо 1 регістр PIN змінюємо стан лінії на протилежний
       while(bar--) // затримка  bar мс
       {
        _delay_ms(1); // бібліотечна функція затримки в 1 мс
       }
// ------------------------------------------------------------------------------------
// ----------------- зміна періоду сигналу ------------------------------------------
       if (Check_Button(Knopka)) // перевіряємо чи натиснута кнопка
       {
         delay_time=delay_time>>2; // зменшуємо період в 2 рази
          if (delay_time<10)// якщо період менше 20 мс, то затримку встановлюємо на максимум
              {    delay_time=MAX_TIME;  }
       }
//-------------------------------------------------------------------------------
    }
}
```

Програма складається із стандартних частин: директиви препроцесора, прототипи функцій, опис функцій та головна частина. Для коректної роботи компілятора, потрібно вказати значення тактової частоти мікроконтролера за допомогою спеціальної константи *F_CPU*, частота вказується в Гц:

```
#define F_CPU 16000000UL.
```

Потім, за допомогою директив препроцесора, підключаються бібліотеки:

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

у біблітеці `<avr/io.h>` описані символьні імена та їх фізичні адреси в пам'яті МК ATmega328P регістрів вводу-виводу, контрольних регістрів та регістрів даних периферійних присроїв МК та назви окремих бітів, які розміщені в цих регістрах. У `<util/delay.h>` описана функція затримки `_delay_ms(N_ms)`,яка реалізує програмну затримку в задану кількість `N_ms` мілісекунд.

Наступні директиви `#define` задають користувацькі назви ПВВ та їх ліній, до яких підключені кнопки і світлодіод. Перевагою користувацьких назв є більш зрозумілий текст програми.

В програмі є дві функції: `Init_Port()` – виконує початкове налаштування ліній портів, до яких підключені кнопка та світлодіод, `Check_Button(Button)` – перевіряє стан лінії порту, до якої підключена кнопка.

Оскільки налаштування ліній портів зводиться до запису 1 чи 0 в певний розряд контрольного регістра, то розглянемо детально як це відбувається за допомогою С. В мові С відсутні прямі програмні засоби (команди чи функції) для маніпуляцій із окремим бітами, тому для цього використовують порозрядні логічні операції. Всього застосовують 4 логічні операції, їх назва, позначення та таблиці істинності наведені в таблиці 1.1.

Таблиця 1.1. Таблиці істиності для логічних операцій.

логічне І (AND)		логічне АБО(OR)		виключаюче АБО (XOR)		інверсія (NOT)	
Операнди	Результат	Операнди	Результат	Операнди	Результат	Операнд	Результат
0 & 0	0	0 | 0	0	0 ^ 0	0	~0	1
0 & 1	0	0 | 1	1	0 ^ 1	1	~1	0
1 & 0	0	1 | 0	1	1 ^ 0	1		
1 & 1	1	1 | 1	1	1 ^ 1	0		

Коли порозрядні логічні операції виконуються над числами, то їх спершу потрібно представити у двійковому форматі, а потім виконати логічну операцію над кожною парою бітів (якщо операція бінарна) чи над кожним бітом (якщо операція унарна).

Окрім того використовуються операції порозрядного зсуву:

>> зсув праворуч - зсуває біти лівого операнда на число розрядів, яке вказано правим операндом. При цьому праві біти будуть втрачені, якщо лівий операнд є ціле беззнакове число, то ліві біти, які звільнилися заповнюються нулями, якщо число із знаком то заповнюються знаком – 1. Математичним еквівалентом операції >> на *n* розрядів є поділ числа на 2^n.

<< зсув ліворуч - зсуває біти лівого операнда на число розрядів, яке вказано правим операндом, при цьому ліві біти втрачаються, а праві заповнюються нулями. Математичним еквівалентом операції << на *n* розрядів є множення числа на 2^n, за умови, якщо старші біти не втрачені.

За допомогою побітових операцій відбуваються маніпуляції із бітами в регістрах МК. Розглянемо найбільш вживані.

Запис 1 в деякий розряд регістру із зануленням решти відбувається за допомогою команди:

```
Register_Name = 1<<Bit_Numer.
```

Наприклад `DDRB = 1<<PB3`, результат послідовного виконання операцій в таблиці:

	Номери розрядів							
	7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
`1`	0	0	0	0	0	0	0	1
`1<<PB3`	0	0	0	0	1	0	0	0
`DDRB = 1<<PB3`	0	0	0	0	1	0	0	0

Якщо потрібно встановити декілька бітів одночасно, то використовують операцію порозрядне АБО:

```
Register_Name = (1<<Bit_Numer1)|(1<<Bit_Numer2).
```

Наприклад: `DDRB = (1<<PB2)|(1<<PB6)`:

Операції	Номери розрядів							
	7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
`1<<PB2`	0	0	0	0	0	1	0	0
`1<<PB6`	0	1	0	0	0	0	0	0
`(1<<PB2)|(1<<PB6)`	0	1	0	0	0	1	0	0
`DDRB = (1<<PB2)|(1<<PB6)`	0	1	0	0	0	1	0	0

Встановлення біта в 1 в певному розряді без зміни решти розрядів регістру:

```
Register_Name |= (1<<Bit_Numer).
```

Наприклад: `DDRB|=(1<<PB4),` або у розширеному записі `DDRB=DDRB|(1<<PB4).`

Операції	Номери розрядів							
	7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
`1<<PB4`	0	0	0	1	0	0	0	0
`DDRB`	0	1	0	0	0	1	0	0
`DDRB | (1<<PB4)`	0	1	0	1	0	1	0	0
`DDRB = DDRB | (1<<PB4)`	0	1	0	1	0	1	0	0

Якщо потрібно встановити декілька бітів, то можна використовувати логічне АБО для об'єднання:

```
Register_Name |= (1<<Bit_Numer1)|(1<<Bit_Numer2)
```

Встановлення нуля в певному розряді без зміни інших розрядів (скидання біта):

```
Register_Name &= ~(1<<Bit_Numer)
```

Наприклад `DDRB &= ~(1<<PB6)`

Операції	Номери розрядів							
	7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
`1<<PB6`	0	1	0	0	0	0	0	0
`~(1<<PB6)`	1	0	1	1	1	1	1	1
`DDRB`	0	1	0	1	0	1	0	0
`DDRB & ~(1<<PB6)`	0	0	0	1	0	1	0	0
`DDRB &= ~(1<<PB6)`	0	0	0	1	0	1	0	0

Інверсію бітів в регістрі виконують за допомогою операції XOR:

```
Register_Name ^= (1<<Bit_Numer1)| (1<<Bit_Numer2)…
```

Наприклад інвертуємо біти `PB2`, `PB5` в `DDRB:`

```
DDRB ^=(1<<PB5)|(1<<PB2)
```

Операції	Номери розрядів							
	7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
`1<<PB5`	0	0	1	0	0	0	0	0
`1<<PB2`	0	0	0	0	0	1	0	0
`(1<<PB5)|(1<<PB2)`	0	0	1	0	0	1	0	0
`DDRB`	0	0	0	1	0	1	0	0
`DDRB ^=(1<<PB5)|(1<<PB2)`	0	0	0	1	0	0	0	0

Для перевірки чи встановлений біт в регістрі використовується команда

```
Register_Name & (1<< Bit_Numer),
```

Якщо біт рівний 0, то результатом виконання такої команди буде 0, а в протилежному випадку не рівний 0. Для того, щоб отримати 1 потрібно зсунути праворуч на `Bit_Numer:`

```
(Register_Name & (1<< Bit_Numer))>> Bit_Numer,
```

таку конструкцію було використано для перевірки стану кнопки:

```
unsigned char foo =( (~Knopka_PIN) & (1<<Button) )>>Button;
```

Значення регістру `Knopka_PIN` інвертується, оскільки, при натисканні кнопки сигнал змінюється від 1 до 0.

Виконання програми відбувається в головній частині програми – функція `int main(void)`, її можна розділити на дві частини: частина яка виконується одноразово,

це частина програми від початку `main(void)` до циклу `while(1)`, та основної частини програми яка буде виконуватися постійно – нескінченний цикл. Як правило, в програмах для МК присутній нескінчений цикл, де виконується функціональна частина програми. Якщо написати програму без нескінченного циклу, то код програми виконається лише один раз і МК перейде в режим енергозбереження, і наступне виконання програми буде можливе, лише після апаратного перезапуску МК: подати низький рівень напруги на лінію Reset.

В першій частині нашої програми виконують початкове налаштування периферійних пристроїв – викликаємо функцію налаштування ПВВ `Init_Port()`, проводимо початкову ініціалізацію змінних та виконуємо однократні дії: включаємо світлодіод та очікуємо натискання кнопки.

В основній частині, в нескінченному циклі, формуємо змінний прямокутний сигнал із заданою частотою за допомогою функції затримки `_delay_ms(1)` в одну мілісекунду. Наступна частина програми перевіряє чи була натиснута кнопка і, якщо була, то зменшує затримку в два рази: `delay_time=delay_time>>1.` Операція ділення на 2 замінена на зсув праворуч. Недоліком програми є те, що всі команди в циклі `while(1)` виконуються послідовно і наявна велика затримка, яка призводить до того, що програма пропускає момент натискання кнопки. Щоб усунути ці недоліки потрібно, щоб МК реагував на натискання кнопки незалежно від виконання основного циклу програми – таку роботу забезпечує механізм зовнішніх переривань (буде розглянуто в лабораторній роботі №2 «Зовнішні переривання»)

Семисегментний індикатор – пристрій призначений для відображення цифрової інформації, який складається із 7 або 8 елементів – сегментів які можуть включатися і виключатися окремо. Сегменти розташовані таким чином щоб за допомогою їх комбінації утворювалися цифри (рис. 5, а). Для підсвітки сегментів використовують світлодіоди, і для зменшення кількості виводів, світлоідоди з'єднують між собою анодами – схема із загальним анодом (СА – *common anodes*) (рис. 5,б) або катодами (СС – *common cathodes*) – схема із загальним катодом (рис. 5,в).

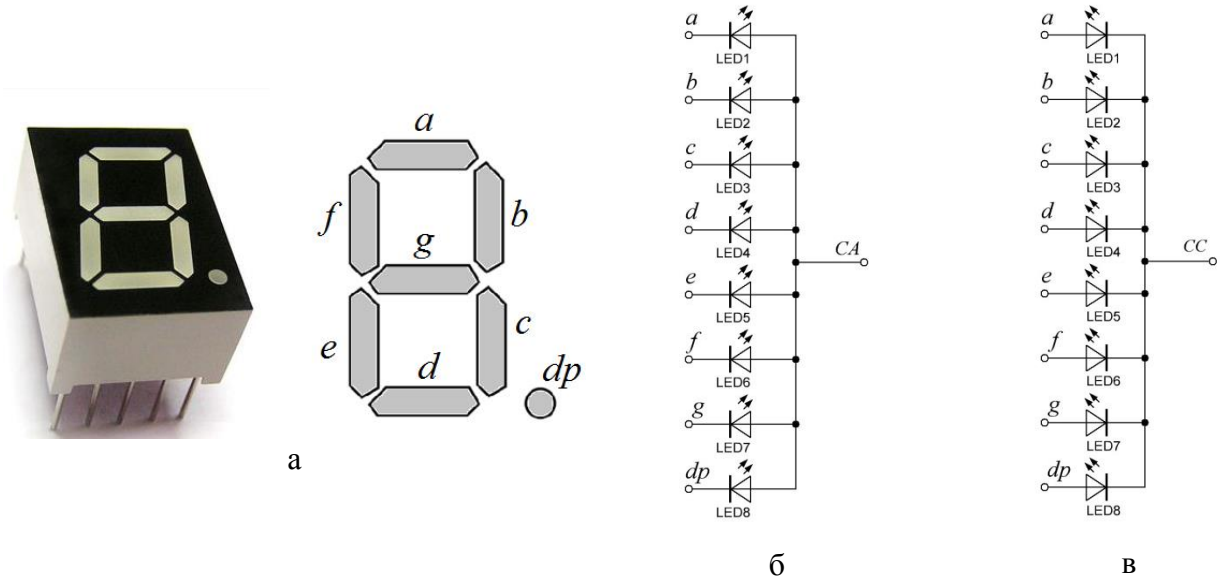


Рис. 5. а) семисегментний індикатор; б) включення із загальним анодом; в) включення із загальним катодом.

При включенні в схему загальний анод підключається до напруги живлення, а загальний катод до землі, відповідно, для включення сегментів в схемі СА потрібно сегмент підключати до землі, а у схемі СС – до напруги живлення. Сегменти прийнято позначати латинськими літерами *a, b, c, d, e, f, g* часто додають восьмий елемент – десяткова кома – *dp*. Для обмеження струму, через світлодіоди семисегментного індикатора послідовно із виводами потрібно підключати обмежуючі резистори. Розглянемо спосіб відображення цифр за допомогою семисегментного індикатора. Підключимо кнопку та семисегментний

індикатор до МК за схемою наведеною на рис. 6. Сформулюємо задачу: будемо підраховувати кількість натискань на кнопку і виводити це число на семисегментний індикатор. Реакція на кнопку буде не по натисканню, коли відбувається зміна сигналу 1→0 (задній фронт сигналу), а по відпусканню кнопки, коли зміна сигналу 0→1 (передній фронт сигналу). При натисканні на кнопку світиться світлодіод, а при відпусканні гасне.

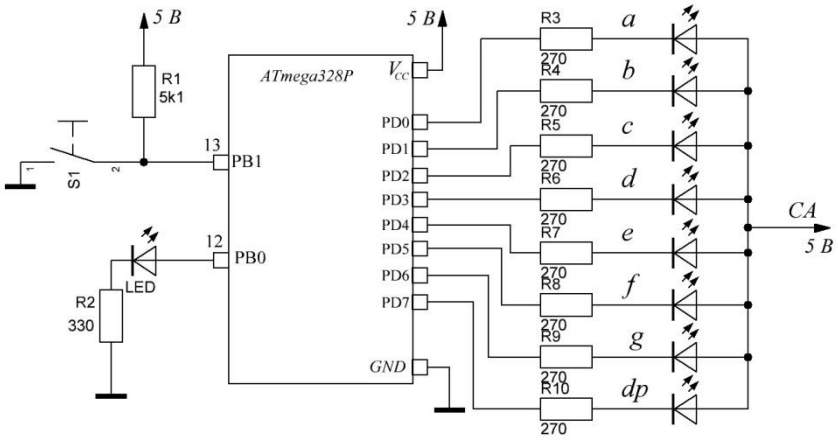


Рис.6. Підключення семисегментного дисплея до МК ATmega328P.

Для того щоб вивести на дисплей цифру 1 потрібно, щоб світилися тільки сегменти *b, c* для цього на лінії порту PORTD PD1, PD2 вивести низький потенціал – логічний 0, а на решту ліній вивести високий потенціал – логічну 1. Відповідно до схеми підключення (рис. 6) для цього PORTD потрібно записати число 249, яке у двійковому форматі: 0b11111001, чи в шіснадцятиричній системі 0xF9. Для кожної цифри потрібно сформувати код, який буде відповідати її зображенню на семисегментному дисплеї. Для цього побудуємо таблицю:

Таблиця 1.2. Семисегментний код для відображення десяткових цифр.

	сегменти	dp	g	f	e	d	c	b	a	HEX	DEC
	PORTD	PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0		
Цифри	0	1	1	0	0	0	0	0	0	0xC0	192
	1	1	1	1	1	1	0	0	1	0xF9	249
	2	1	0	1	0	0	1	0	0	0xA4	164
	3	1	0	1	1	0	0	0	0	0xB0	176
	4	1	0	0	1	1	0	0	1	0x99	153
	5	1	0	0	1	0	0	1	0	0x92	146
	6	1	0	0	0	0	0	1	0	0x82	130
	7	1	1	1	1	1	0	0	0	0xF8	248
	8	1	0	0	0	0	0	0	0	0x80	128
	9	1	0	0	1	0	0	0	0	0x90	144
відсутнє зображення		1	1	1	1	1	1	1	1	0xFF	255

за допомогою якої знайдемо семисегментний код. Цей код помістимо в масив таким чином, щоб номер елементу масиву відповідав його коду:

```
unsigned char Digits[11]={0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8, 0x80, 0x90, 0xFF};
```

Цей масив буде використаний в програмі для перекодування цифри в зображення. Програма, яка реалізує цю задачу, наведена в лістингу 1.2. Логіку роботи програми можна зрозуміти із коментарів.

Лістинг 1.2. Відображення цифр за допомогою семисегментного світлодіодного дисплея

```
/* ****************************************************************
/  Програма, яка ставить у відповідність натуральному числу (кількість подій - натискання
/  кнопки)  код семисегментного дисплея - графічне зображення цифри
/
/        7-сегментний світллодіодний дисплей приєднаний до S7LED_Port:
/   S7LED_Port's bit     |7 |6|5|4|3|2|1|0|
/   7 seg LED            |dp|g|f|e|d|c|b|a|
/
/  7-сегментний світллодіодний дисплей із загальним анодом -
/  сегменти будуть світитися, якщо рівень напруги - 0.
/ ************************************************************* */
#define F_CPU 16000000UL

#include <avr/io.h>
#include <util/delay.h>

#define LED_Button_PORT PORTB  // порт, до якого буде приєднано світлодіод та кнопка
#define LED_Button_DDR  DDRB   //  регістр вибору напрямку роботи порта світлодіода
#define Button_PIN PINB        // регістр для перевірки стану кнопки
#define LED PB0                // світлодіод приєднаний до PB0
#define Kn_pin PB1             // кнопка приєднана до PB1

#define S7LED_Port PORTD    // 7-сегментний дисплей приєднаний до PORTD
#define S7LED_DDR DDRD      // регістр задає режим роботи (вхід-вихід) LED_7seg_Port

void Int_Port(void);              // початкова ініціалізація портів
unsigned char Chk_Kn(char k);     // перевірка стану кнопки

// кодування цифр в 7-сег. код:
unsigned char Digits[11]={0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8, 0x80, 0x90, 0xFF};

//---------- Опис функцій -------------------------
void Int_Port(void)
{
// лінія кнопки - вхід, підключаємо підтяюгуючий резистор,
// якщо не підключити, то розрив в колі кнопки буде сприйматися як натиснута кнопка
       LED_Button_DDR &=~(1<<Kn_pin);
       LED_Button_PORT |=(1<<Kn_pin);

       LED_Button_DDR |=(1<<LED);       // лінія світлодіоду - вихід
       LED_Button_PORT &=~(1<<LED);     // світлодіод виключаємо

       S7LED_DDR=0xFF;     // порт до якого підключений 7-ми сег. дисплей - вихід
       S7LED_Port=0xFF;     // всі лінії порту  1 - дисплей вимкнений
}
// фукція перевірки стану кнопки, повертає 1, якщо кнопка натиснута,
// але спрацьовує по відпускнню кнопки
 unsigned char Chk_Kn(char k)
  {
       volatile char Kn=0;
       if ( ((~(Button_PIN)) & (1<<k)) )   // перевіряємо чи натиснута кнопка
       {
              LED_Button_PORT |=(1<<LED);    // включаємо світлодіод
             _delay_ms(10);                    // боремося із "брязкотом" контактів  кнопки
             Kn=((~(Button_PIN))&(1<<k))>>k;   // ще раз перевіряємо стан кнопки

           while((~Button_PIN)&(1<<k))       // чекаємо на відпускання кнопки
            {
               _delay_ms(1);
            }
       }
       LED_Button_PORT &=~(1<<LED);         // виключаємо світлодіод
       return Kn;
  }
int main(void)
{
```

```
        Int_Port();
        // включаємо по черзі сегменти
        char j=7;
        while(j--)
         {
                S7LED_Port=~(1<<(7-j));
              _delay_ms(300);
         }

        S7LED_Port=Digits[0]; // кнопку не натискали - на дисплей виводимо код нуля.
        unsigned char n=0; // лічильник кількості натискань кнопки

    while (1)
    {
       volatile unsigned char f=Chk_Kn(Kn_pin);  // перевіряємо стан кнопки
       if(f)
        {
          // якщо лічильник перевищує 11, то його зануляємо - у нас лише 1 розряд
         if (++n==11) n=0;
         S7LED_Port=Digits[n]; // вивели в LED_7seg_Port код, що відповідає поточному n
        }
    }
}
```

Завдання до лабораторної роботи.

1. Створити проект в Atmel Studio 7, надрукувати програму наведену в лістинг 1.1. Вивчити логіку роботи програми за допомогою симулятора.
2. Створити проект в Atmel Studio 7, надрукувати програму наведену в лістинг 1.2. Вивчити логіку роботи програми за допомогою симулятора.

Розробити алгоритм, написати програму, відладити роботу за допомогою симулятора та перевірити за допомогою стенда:

3. «Біжучі вогні» - світлодіоди засвічуються по черзі із однаковим інтервалом в 1 с. При натисканні на кнопку 1 – інтервал зменшується в 2 рази. При натисканні на кнопку 2 «біжучі вогні» перетворюються на «біжучу тінь» - всі світлодіоди світяться за винятком одного №1, потім всі світяться за винятком №2 …. Використати кнопки та світлодіодну лінійку.
4. Написати програму, яка імітує роботу світлофора для пішоходів: при натисканні на кнопку через 30 секунд червоне світло «моргає» 3 рази, потім вмикається жовте світло і через 1 секунду вимикається жовте і вмикається зелене світло на 10 с. На 7 сегментному індикаторі виконати зворотній відлік 10-ти секунд. Після закінчення 10 сек на 1 сек увімкнути жовте світло, а потім включити червоний.
5. Написати програму, яка виводить на 7 сегментний дисплей номер кнопки, яка була натиснута. При натисканні кнопки виводиться її номер (1-6) номер на 5 сек, а потім виводиться 0.
6. «Перетворювач кодів». Написати програму яка перетворює код: кількість подій (натискання кнопки - натуральне число) → число-імпульсний код (кількість імпульсів за проміжок часу) → бінарний код (виводиться на 8 світлодіодів) → 7-ми сегментний код відповідного числа. Максимальна кількість подій – 19. Старший розряд – десятки відображати за допомого світлодіоду, молодший розряд за допомогою 1-го розряду 7 сегментного індикатора (включається перемикачем біля індикатора– див. опис плати).

Контрольні запитання:

1. Паралельні порти вводу-виводу: призначення, будова та електричні характеристики.
2. Наявні порти в ATmega328P.
3. Регістри паралельних портів, їх назви та призначення. Нумерація та позначення бітів в регістрах.

4. Режими роботи паралельних портів, методика налаштування ліній портів на вхід та на вихід.
5. Основні схеми підключення кнопок до портів мікроконтролера. Методика зчитування сигналів від механічних кнопок та перемикачів.
6. Основні схеми підключення світлодіодів до портів мікроконтролера.
7. Будова семисегментного світлодіодного дисплея. Методика відображення цифр за допомогою дисплея.

Література

1. Datasheet ATmega328P. http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf
2. Программирование микроконтроллеров ATMEL на языке С. Прокопенко В.С., К.: «МК-Пресс», 2012. – 320с.
3. Програмирование на языке С для AVR и PIC микроконтролеров. Шпак Ю.А., К.: «МК-Пресс», 2016. – 544стр.
4. Микроконтроллеры AVR семейств Tiny и Mega фирмы ATMEL. – М.: Издательский дом «Додэка-XXI», 2004. – 560 с.
5. Application Note_1497 AVR035: Efficient C Coding for 8-bit AVR microcontrollers. http://ww1.microchip.com/downloads/en/AppNotes/doc1497.pdf
6. Application Note_AVR42787 AVR Software User Guide http://ww1.microchip.com/downloads/en/AppNotes/Atmel-42787-AVR-Software-User-Guide_ApplicationNote_AVR42787.pdf

Лабораторна робота 2.

Вивчення зовнішніх переривань Atmega328P

Мета: Вивчення зовнішніх переривань ATmega328P. Вивчення методів налаштування та використання зовнішніх переривань.

Основне призначення мікроконтролерних систем управління – це є реакція на певні зовнішні чи внутрішні події. Подіями можуть бути зовнішні сигнали, які поступають на лінії портів введення-виведення, або внутрішні сигнали, які генеруються в периферійних пристроях МК. Наприклад: завершилося аналогово-цифрове перетворення, закінчився відлік таймера, тощо. У більшості випадків зовнішні події не синхронізовані із роботою МК і можуть наставати в довільний момент часу. Для того, щоб не пропустити подію, використовують два основні методи: постійна чи періодична перевірка наявності події в нескінченному циклі (опитування кнопки лістинг 1.1 лабораторна робота 1), або використання апаратних переривань. При використанні 1-го способу програма організована у вигляді нескінченного циклу де функції, які виконуються МК виконуються послідовно одна за одною, і якщо одна із функцій буде очікувати подію, то інші функції в цей час не виконуються. Окрім того є ризик, що подія не настане і МК залишиться в стані очікування, а якщо потрібно відслідкувати настання декількох подій, то існує ризик пропустити інші події.

Вирішити цю проблему можна за допомого механізму апаратних переривань. ***Переривання*** – це метод реакції МК на подію, коли нормальне (послідовне) виконання програми, при настанні події, призупиняється і подальше виконання програми передається спеціальній функції – обробнику переривання (Interrupt Handler or Interrupt Service Routine - ISP), яка формує реакцію системи на зовнішню подію. Після закінчення роботи обробника переривань, МК продовжує виконання основної (фонової) програми із того місця, де відбулося переривання. Виконання переривань забезпечується апаратно, до кожного із них прив'язана унікальна комірка пам'яті в області пам'яті програм. В цю комірку поміщається адреса функції обробника переривання – ці комірки називають ***векторами переривань***. При настанні події, яка викликає переривання, до виконання завантажується не наступна команда програми, а команда, яка знаходиться в комірці-вектор переривання. Після цього відбувається перехід до виконання функції – обробника переривання.

Детальний опис переривань наведений в технічному описі (таблиця 16-1. Вектори переривань в ATmega328P. ст. 82). В ATmega328P налічується 26 зовнішніх та внутрішніх переривань, таблиця векторів переривань, як правило, розміщена на початку пам'яті програм, починаючи із адреси 0x0000 – вектор Reset. Переривання мають різний пріоритет до виконання. Переривання із меншою адресою мають вищий пріоритет, це означає, що якщо одночасно виникає запит на обробку декількох переривань, то буде виконуватися те, яке має вищий пріоритет. Розглянемо частину, яка стосується зовнішніх переривань. Наступні 5 векторів відносяться до зовнішніх переривань і називаються **INT0**, **INT1** (INT – скорочення від Interrupt) та **PCINT0**, **PCINT1**, **PCINT2** (Pin Change Interrupt). Ці переривання викликаються зовнішніми подіями (рис.1).

16.1. Interrupt Vectors in ATmega328/P

Table 16-1. Reset and Interrupt Vectors in ATmega328/P

Vector No	Program Address(2)	Source	Interrupts definition
1	0x0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 0
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2

Рис.1. Таблиця векторів переривань (частина, яка описує зовнішні переривання) ATmega328P

Зовнішні переривання є індивідуальними - **INT0**, **INT1,** або груповими та **PCINT0**, **PCINT1**, **PCINT2**. Індивідуальні переривання генеруються, якщо настає подія на одній лапці МК: для **INT0** призначена лапка №32 PD2, а для **INT1** лапка №1 PD3 (рис. 1 лабораторна робота 1). Групове переривання буде генеруватися, якщо подія виникає на будь-якій одній лапці із групи, яка призначена для зовнішнього переривання, наприклад до **PCINT0** відносяться лапки які позначені, як PCINT0, PCINT1, … PCINT7 (лапки №№12 – 17, 7, 8, які відносяться до PORTB), для **PCINT1** призначені лапки: PCINT8 – PCINT14 (PORTC, лапки №№ 23 – 29), для **PCINT2** призначені лапки: PCINT16 – PCINT23 (PORTD, лапки №№ 30 – 32, 1, 2, 9 – 11).

Оскільки лінії введення-виведення МК оперують із цифровими сигналами «0» та «1», яким відповідають певні рівні напруги, то зовнішніми подіями будуть наявність чи зміна цифрового сигналу на відповідній лапці МК.

Такими подіями будуть (рис. 2):

- передній фронт (зростання) сигналу – зміна від V_0 до V_1;
- задній фронт (спад) сигналу – зміна від V_1 до V_0;
- низький рівень сигналу.

Такі події можуть бути отримані за допомогою звичайної механічної кнопки.

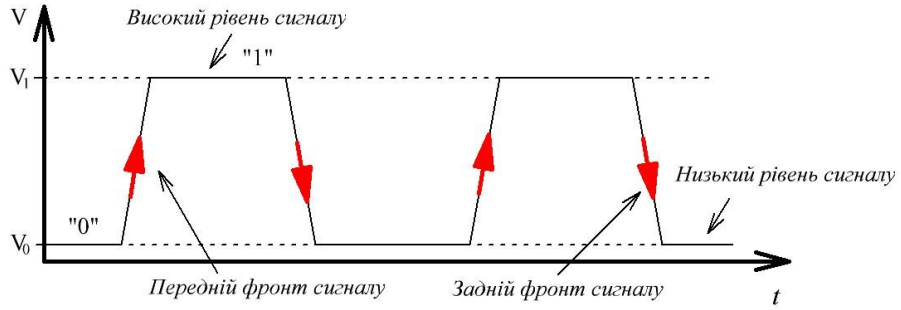


Рис. 2. Види подій, які викликають зовнішні переривання. V_1 - напруга яка відповідає логічної одиниці (високий рівень сигналу), V_0 - напруга логічного нуля (низький рівень)

Зовнішні переривання **INT0**, **INT1** можуть бути згенеровано за переднім, заднім фронтом сигналу чи за наявності низького рівня на відповідних лініях введення-виведення. Переривання за фронтами сигналу є синхронним із системною частотою МК, отже, для надійної реєстрації фронту сигналу потрібно, щоб зовнішній імпульс тривав більше за період системної частоти МК. Переривання за низьким рівнем є асинхронним – може використовувати для виведення МК із режиму енергозбереження. Якщо переривання активуються низьким рівнем, то запит на переривання буде сформований після закінчення виконання поточної команди. Переривання за низького рівня сигналу буде відбуватися доти, поки такий сигнал присутній на відповідній лінії введення-виведення.

Переривання **PCINT0**, **PCINT1**, **PCINT2** є асинхронними і генеруються за будь-якої зміни (передній за задні фронти) сигналу на відповідних лініях введення-виведення.

Управління та налаштування зовнішніх переривань в ATmega328P.

Налаштування та управління зовнішніми перериваннями відбувається шляхом встановлення необхідних бітів в управляючих та контрольних регістрах. Всі переривання, не тільки зовнішні, мають 2-х ступеневу систему дозволу – глобальну та локальну. Глобальний дозвіл дозволяє/забороняє одночасно всі переривання. В регістрі стану ***SREG*** є біт, який називається I (Global Interrupt Enabled). Для встановлення чи скидання цього біта є спеціальні функції, які описані в бібліотечному файлі *<avr/interrupt.h>*

```
cli();    очистити біт I=0 – заборонити переривання
sei();    встановити біт I=1 – дозволити переривання
```

Для локального дозволу переривань існують спеціальні контрольні регістри наявні для кожного переривання із своєю унікальною назвою та розташуванням. Для зовнішніх переривань **INT0**, **INT1** цей регістр називається **EIMSK** (External Interrupt Mask Register):

bit	7	6	5	4	3	2	1	0
EIMSK	-	-	-	-	-	-	**INT1**	**INT0**

В цьому регістрі лише 2 робочих біти, які називаються **INT1, INT0**, які відповідають за локальний дозвіл зовнішніх переривань **INT0**, **INT1.** Якщо біт **INT1** чи **INT0** встановлений і I=1 в ***SREG,*** то відповідні зовнішні переривання будуть дозволені.

Вид подій, на які буде реагувати зовнішні переривання вибираються шляхом конфігурації регістру **EICRA** (External Interrupt Control Register A)

bit	7	6	5	4	3	2	1	0
EICRA	-	-	-	-	**ISC11**	**ISC10**	**ISC01**	**ISC00**

Біти **ISC11, ISC10** будуть визначати вид події (передній, задній фронт чи низький рівень сигналу) для **INT1**, а **ISC01, ISC00** для **INT0** відповідно. Конфігурація бітів для кожної події описана в таблиці 1.

Таблиця 1. Конфігурація бітів регістру **EICRA.**

Значення бітів		Опис події
ISC11/ ISC01	**ISC10/ ISC00**	
0	0	низький рівень сигналу – логічний «0» буде генерувати переривання **INT1/ INT0**
0	1	будь-яка зміна логічного сигналу: передній («0» → «1») чи задній фронт («1» → «0») буде генерувати переривання **INT1/ INT0**
1	0	задній фронт сигналу («1» → «0») буде генерувати запит на переривання **INT1/ INT0**
1	1	передній фронт сигналу («0» → «1») буде генерувати запит на переривання **INT1/ INT0**

Коли логічний «0» чи зміна логічного стану присутня на лінії введення-виведення згенерує запит на переривання, то в регістрі **EIFR** (External Interrupt Flag Register – регістр прапорців зовнішніх переривань) буде встановлений спеціальний біт **INTF1** чи **INTF0** – прапорець, який буде сигналізувати що відбувся запит на переривання.

bit	7	6	5	4	3	2	1	0
EIFR	-	-	-	-	-	-	**INTF1**	**INTF0**

Якщо переривання дозволені, то відбудеться перехід виконання програми на відповідний вектор переривання: якщо встановлено **INTF0,** то перехід на **INT0,** а якщо **INTF1** то на **INT1.** Вектор переривань є коміркою програмної пам'яті в якій записана команда переходу до відповідної підпрограми-обробника, яка буде реакцією на переривання. Ця підпрограма оголошується за допомогою спеціального макросу `ISR(vector).` Під час виконання підпрограми-обробника всі переривання автоматично забороняються глобально (I=1). Якщо потрібно і цей час дозволити переривання, то це виконується програмно за допомогою функції `sei();.` Після закінчення виконання підпрограми, прапорці **INTF1** чи **INTF0** будуть скинуті і МК повернеться до виконання основної програми. Прапорці можуть бути скинуті програмно, якщо в ці біти записати логічні «1», наприклад скинемо **INTF1**:

```
EIFR |=(1<< INTF1);.
```

Якщо зовнішні переривання заборонені, то прапорці встановлюються, але перехід до вектору переривань не відбувається. В цьому випадку, прапорці можна скидати програмно.

Зовнішні переривання **PCINT0**, **PCINT1**, **PCINT2** мають власну групу контрольних регістрів: PCICR (Pin Change Interrupt Control Register), PCIFR (Pin Change Interrupt Flag Register), PCMSK0, PCMSK1, PCMSK2 (Pin Change Mask Register).

Регістр PCICR (Pin Change Interrupt Control Register) має 3 управляючі біти: PCIE0, PCIE1, PCIE2 (Pin Change Interrupt Enable) кожен з яких відповідає за свою групу переривань, якщо встановлений біт I в SREG та встановлені біти:
PCIE0 – зовнішнє переривання PCINT0 дозволене і буде виконуватися із вектору PCIE0;
PCIE1 – зовнішнє переривання PCINT1 дозволене і буде виконуватися із вектору PCIE1;
PCIE2 – зовнішнє переривання PCINT2 дозволене і буде виконуватися із вектору PCIE2.

bit	7	6	5	4	3	2	1	0
PCICR	-	-	-	-	-	**PCIE2**	**PCIE1**	**PCIE0**

Регістр PCIFR (Pin Change Interrupt Flag Register) містить 3 біти PCIF0, PCIF1, PCIF2:

bit	7	6	5	4	3	2	1	0
PCIFR	-	-	-	-	-	**PCIF2**	**PCIF1**	**PCIF0**

Якщо відбудеться переривання PCINT0, PCINT1, PCINT2, то буде встановлений відповідний йому прапорець PCIF0, PCIF1, PCIF2. Після обробки переривання прапорець скидається в 0 автоматично, або це можна виконати програмно записавши в нього «1».
Регістри PCMSK0, PCMSK1, PCMSK2 (Pin Change Mask Register) призначені для індивідуального управління кожною лінією портів введення-виведення, які будуть генерувати зовнішні переривання PCINT[2..0].

bit	7	6	5	4	3	2	1	0
PCMSK0	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0

bit	7	6	5	4	3	2	1	0
PCMSK1	-	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8

bit	7	6	5	4	3	2	1	0
PCMSK2	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16

Кожен із цих бітів PCINT[7:0], PCINT[14:8], PCINT[23:16] – визначає чи буде дозволене переривання за зміною цифрового сигналу на відповідній лапці введення-виведення. Якщо біт PCINT[7:0], PCINT[14:8], PCINT[23:16] встановлений в «1» і біти PCIE0, PCIE1, PCIE2 встановлені в PCICR, то переривання буде дозволене на відповідній лапці введення-виведення. Якщо біт PCINT[7:0], PCINT[14:8], PCINT[23:16] скинутий в «0», то зміна цифрового сигналу на лапці введення-виведення не викликатиме запит на переривання.

Переривання PCINT0, PCINT1, PCINT2 є груповими – якщо декілька ліній введення-виведення можуть викликати переривання, то для визначення, яка лінія введення-виведення згенерувала переривання, потрібно в функції обробки переривання прочитати значення відповідного регістру PINx.

Використання зовнішніх переривань можна описати таким алгоритмом:

- вибираємо тип події: передній, задній фронт чи низький рівень сигналу;
- проводимо налаштування відповідної лінії введення-виведення на вхід, вибираємо тип події, включаємо потрібне переривання локально;
- створюємо функцію – реакцію на переривання яка оголошується:

```
ISR(INT0_vect) // для прикладу візьмемо INT0
  {
  // код який буде реакцією на переривання:
  }
```

- в основній програмі дозволяємо переривання за допомогою функції `sei();`

Розглянемо приклад використання зовнішніх переривань. Змінимо програму наведену в лістингу 1.1 (Лабораторна робота №1 «Вивчення роботи паралельних портів Atmega328P») таким чином, щоб для опитування кнопки було використане зовнішнє переривання **INT1. INT1** буде активуватися заднім фронтом сигналу, оскільки кнопки лабораторного стенду мають підтягуючі резистори до напруги живлення (кнопка S2 на рис. 3. Лабораторна робота

№1 «Вивчення роботи паралельних портів Atmega328P»). Програмний код наведений в лістингу 2.1.

Лістинг 2.1. Використання зовнішнього переривання **INT1** для опитування кнопки.

```
/*
 * lab2-1.c
 * опитування кнопки із використанням
 * зовнішнього переривання INT1
 */

#define F_CPU 16000000UL //тактова частота роботи МК
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>  // бібліотека де описані переривання

// Кнопку підключаємо до PORTD
#define Knopka_PORT PORTD
#define Knopka_DDR DDRD
#define Knopka_PIN PIND
// кнопку підключаємо до PD3 – лапка, яка призначена для переривання INT1
#define Knopka PD3

// Світлодіод підключаємо до PORTD
#define LED_PORT PORTD
#define LED_DDR DDRD
#define LED_PIN PIND
#define LED_1 PD5  // світлодіод - PD5 - вихід
#define MAX_TIME 1000 // максимальний період 2*MAX_TIME

/* оголошуємо змінну Button_F, яка буде прапорцем натиснення кнопки
 її оголошуємо із кваліфікатором volatile - це сигнал компілятору що ця змінна
 може змінюватися апаратним методом (під час обробки переривання INT1)
 незалежно від тексту програми, її не потрібно оптимізовувати */
volatile unsigned char Button_F=0;

//---------------- Прототипи функцій ----------------------------------------------------
void Init_Port(void); // Функція початкового налаштування ліній вводу виводу портів
void Init_Ext_Interrupt(void); // Функція початкового переривання INT1

//--------- Опис функцій -------------------------------------
void Init_Port(void)
{
	Knopka_DDR &=~(1<<Knopka); // кнопку підключаємо до PD3 - вхід
	LED_DDR |=(1<<LED_1); // світлодіод - PD5 - вихід
}

void Init_Ext_Interrupt()
{
  EICRA |= (1<<ISC11); // переривання активується за заднім фронтом: ISC11=1; ISC10=0
  EIMSK |=(1<<INT1);   // активуємо переривання локально
  EIFR |= (1<<INTF1);  // скидаємо прапорець переривання
}

ISR(INT1_vect) // Функція - реакція на переривання
{
  Button_F=( (~Knopka_PIN) & (1<<Knopka) )>>Knopka; //перевіряємо чи була натиснута кнопка
  EIFR |= (1<<INTF1);  // скидаємо прапорець переривання
}
int main(void)
{
	Init_Port();
	Init_Ext_Interrupt();
	sei(); // включаємо переривання
	unsigned int delay_time=MAX_TIME;

	LED_PORT|= (1<<LED_1); // включаємо світлодіод
	while(!Button_F) // чекаємо допоки буде натиснута кнопка
```

```
       Button_F=0; // обнуляємо прапорець
       LED_PORT &= ~(1<<LED_1); // виключаємо світлодіод

    while (1) // головний цикл програми
    {
 // --------------- створюємо періодичний сигнал ----------------------------------------
       unsigned int bar = delay_time;
       LED_PIN |= (1<<LED_1); // записуємо 1 регістр PIN змінюємо стан лінії на протилежний
       while(bar--) // затримка bar мс
       {
       _delay_ms(1);// бібліотечна функція затримки в 1 мс описана в бібліотеці util/delay.h
       }
// ----------------- зміна періоду сигналу ------------------------------------------------
       if (Button_F) // перевіряємо чи натиснута кнопка
       {
        delay_time=delay_time>>1;  // зменшуємо період в 2 рази
        if (delay_time<10) //якщо період менше 20 мс, то затримку встановлюємо на максимум
         {
           delay_time=MAX_TIME;
         }
         Button_F=0; // обнуляємо прапорець     кнопки
        }
//-------------------------------------------------------------------------------
    }
}
```

В цій програмі, для використання переривань, була використана бібліотека `<avr/interrupt.h>`, для ініціалізації зовнішнього переривання використана функція `Init_Ext_Interrupt()`. Для передачі інформації про подію (натиснення кнопки), було використано глобальну змінну `Button_F`. Ця змінна оголошується із спеціальним кваліфікатором `volatile`:

```
volatile unsigned char Button_F=0;
```

який вказує компілятору, що ця змінна може змінюватися незалежно від програмного коду (апаратно) і що її заборонено вилучати при компіляції та оптимізації машинного коду програми. Якщо опустити кваліфікатор `volatile`, то при оптимізації машинного коду ця змінна може бути вилучена і робота програми буде не коректною. Реакцією на переривання є функція, яка оголошується за допомогою спеціального макроса `ISR(vector)`, який описаний в `<avr/interrupt.h>`, де *vector* – це ім'я вектору переривання. Оскільки під час виконання функції `ISR(vector)`, всі переривання автоматично забороняються то існує декілька рекомендацій до написання функцій `ISR(vector)`:

- функції обробки переривань повинні мати мінімально можливий розмір;
- в `ISR(vector)` не повинно бути програмних затримок типу `_delay_ms(1);`
- не повинно бути викликів інших функцій всередині функції обробки переривань.

Приклад використання переривання PCINT1 для опитування кнопок наведено в лістингу 2.2.

Лістинг 2.2. Використання PCINT1 для опитування кнопок

```
/*
 * lab2-2.c
 ************************************************************
/        7-сегментний світллодіодний дисплей приєднаний до S7LED_Port:
/   S7LED_Port's bit     |7 |6|5|4|3|2|1|0|
/   7 seg LED            |dp|g|f|e|d|c|b|a|
/
/  7-сегментний світллодіодний дисплей із загальним анодом -
/  сегменти будуть світитися якщо рівень напруги - 0.
/ ********************************************************** */

#define F_CPU 16000000UL

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

```
#define LED_Button_PORT PORTC // порт, до якого буде приєднано світлодіод та кнопка
#define LED_Button_DDR  DDRC  // регістр вибору напрямку роботи порту світлодіода
#define Button_PIN PINC       // регістр для перевірки стану кнопки

// кнопки приєднані: до PC0 збільшує число, до PC1 зменшує число на дисплеї
#define Kn_Up_pin PC0      // PCINT8
#define Kn_Down_pin PC1    // PCINT9

#define S7LED_Port PORTD    // 7-сегментний дисплей приєднаний до PORTD
#define S7LED_DDR DDRD      // регістр задає режим роботи (вхід-вихід) LED_7seg_Port

void Int_Port(void);      // початкова ініціалізація портів
void PCINT1_Init(void);

// Глобальні змінні
// кодування цифр в 7-сег. код:
unsigned char Digits[11]={0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8, 0x80, 0x90, 0xFF};

/*   змінна, в якій буде зберігатися значення після опитування кнопки
     3 - кнопка не натискалася, 1 - "-" 2 - "+"
     Knopka_F буде змінювати значення в перериванні, тому вона оголошена як volatile */
    volatile unsigned char Knopka_F=0;

//---------- Опис функцій -------------------------
 void Int_Port(void)
{
/* лінія кнопки - вхід, підключаємо підтяжучий резистор, якщо не підключити
  то розрив в колі кнопки буде читатися як натиснута кнопка */
       LED_Button_DDR &= ~( (1<<Kn_Up_pin)|(1<<Kn_Down_pin) );
       LED_Button_PORT |= (1<<Kn_Up_pin)|(1<<Kn_Down_pin);

       S7LED_DDR = 0xFF;     // порт, до якого підключений 7-ми сег. дисплей - вихід
       S7LED_Port = 0xFF;     // всі лінії порту  1 - дисплей вимкнений
}

// налаштування переривання PCINT1
void PCINT1_Init()
{
  PCICR |= (1<<PCIE1);  // дозволяємо переривання PCINT1

// дозволяємо генерувати переривання лапкам, до яких підключені кнопки
  PCMSK1 |= (1<<PCINT9)|(1<<PCINT8);
  PCIFR |= (1<<PCIF1); // скидаємо прапорець переривання PCINT1
}

ISR(PCINT1_vect)
{
  Knopka_F = PINC & ( (1<<Kn_Down_pin)|(1<<Kn_Up_pin) ); // читаємо стан кнопок
  PCIFR |= (1<<PCIF1); // скидаємо прапорець переривання PCINT1
}

int main(void)
{
 Int_Port();
 PCINT1_Init();
 sei();  // дозволяємо глобальні переривання
 // включаємо по черзі сегменти
       char j = 7;
       while(j--)
       {
        S7LED_Port = ~(1<<(7-j));
        _delay_ms(300);
       }

       S7LED_Port = Digits[0]; // кнопку не натискали - на дисплей виводимо код нуля.
       unsigned char n = 0;    // лічильник кількості натискань кнопки
```

```
    while (1)
     {
       if (Knopka_F == 2 && n<9) // збільшення числа
        {
         Knopka_F = 0;      // скидаємо прапорець кнопки
          n++;
        }
       if (Knopka_F == 1 && n>0) // зменшення числа
       {
         Knopka_F = 0;      // скидаємо прапорець кнопки
         n--;
       }
       S7LED_Port=Digits[n]; // число виводимо на дисплей
// вводимо затримку, щоб встигнути побачити цифру на екрані та уникнути брязкоту контактів
    _delay_ms(30);
       }
}
```

Завдання до лабораторної роботи.

1. Створити проект в Atmel Studio 7, надрукувати програму наведену в лістинг 2.1. Вивчити логіку роботи програми за допомогою симулятора.
2. Створити проект в Atmel Studio 7, надрукувати програму наведену в лістинг 2.2. Вивчити логіку роботи програми за допомогою симулятора.

Розробити алгоритм, написати програму, відладити роботу за допомогою симулятора та перевірити за допомогою лабораторного стенду:

3. «Біжучі вогні» - світлодіоди засвічуються по черзі із однаковим інтервалом в 1 с. При натисканні на кнопку 1 інтервал зменшується в 2 рази. При натисканні на кнопку 2 «біжучі вогні» перетворюються на «біжучу тінь» - всі світлодіоди світяться за винятком одного №1, потім всі світяться за винятком №2 …. Використати кнопки та світлодіодну лінійку. Для опитування кнопок використати зовнішні переривання INT0, INT1.
4. Написати програму, яка виводить на 7 сегментний дисплей номер кнопки яка була натиснута. При натисканні кнопки виводиться її номер (1-6). Для опитування кнопок використати зовнішнє переривання PCINT1.

Контрольні запитання:

1. Поясніть що таке переривання та механізм їх роботи.
2. Типи зовнішніх переривань ATmega328P. Які події можуть викликати зовнішні переривання?
3. Управляючі і контрольні регістри зовнішніх переривань INT0, INT1. Методика налаштування зовнішніх переривань INT0, INT1.
4. Управляючі і контрольні регістри зовнішніх переривань PCINT0, PCINT1, PCINT2. Методика налаштування зовнішніх переривань PCINT0, PCINT1, PCINT2.
5. Особливості написання програмного коду функції обробника переривань.
6. Для чого використовується кваліфікатор *volatile* при оголошенні змінних?

Література

1. Datasheet ATmega328P. http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf
2. Программирование микроконтроллеров ATMEL на языке С. Прокопенко В.С., К.: «МК-Пресс», 2012. – 320с.
3. Програмирование на языке С для AVR и PIC микроконтролеров. Шпак Ю.А., К.: «МК-Пресс», 2016. – 544стр.
4. Микроконтроллеры AVR семейств Tiny и Mega фирмы ATMEL. – М.: Издательский дом «Додэка-XXI», 2004. – 560 с.
5. Application Note AN_8468 AVR1200: Using External Interrupts for megaAVR Devices http://www.microchip.com//wwwAppNotes/AppNotes.aspx?appnote=en590935

Лабораторна робота 3.

Вивчення режимів роботи вбудованих таймерів-лічильників в *Atmega328P*.

Мета: навчитися використовувати вбудовані в ATmega328P таймери-лічильники для генерації цифрових та аналогових сигналів, відліку часових інтервалів, та вимірювання частоти.

Таймер-лічильник (*timer-counter*) – це периферійний пристрій в МК, який відміряє наперед задані інтервали часу, підраховуючи кількість електричних імпульсів, які прийшли на його вхід. Імпульси можуть формуватися з системної тактової частоти МК за допомогою вбудованого подільника частоти, або подаватися зовні на спеціальний вивід МК. Після закінчення заданого інтервалу часу таймер-лічильник створює подію – генерує внутрішнє переривання, або створює управляючі сигнали змінюючи стан ліній введення-виведення МК. Таймери можна використовувати для формування точних часових інтервалів, генерації сигналів різної частоти та заповнення (прогальності), послідовності імпульсів заданої довжини.

Основою таймера-лічильника є цифровий лічильник в регістрі якого накопичується число, яке відповідає кількості вхідних імпульсів. Цифровий лічильник доповнений спеціальними модулями, які можуть генерувати події чи сигнали (пристрій порівняння, генератор ШІМ, тощо). Максимальна кількість імпульсів, які може підрахувати таймер-лічильник залежить від розрядності регістру лічильника, вони бувають 8-ми, або 16-ти розрядні. Максимальна кількість імпульсів або максимальне число, яке може зберігатися в 8-розрядному регістрі $MAX = 2^8 - 1 = 255 = 0x FF$, а в 16-розрядному регістрі $MAX = 2^{16} - 1 = 65535 = 0xFFFF$. Мінімальний інтервал часу, який може відрахувати таймер-лічильник, залежить від мінімальної тривалості вхідного імпульсу.

В ATmega328P є наявні 3 таймери-лічильники *TC0* (Timer/Counter0), *TC1*, *TC2* два із них *TC0, TC2* є 8-розрядні, а *TC1* 16-ти розрядний. Таймери *TC0, TC1* можуть використовувати спільний 10-ти розрядний подільник частоти, а *TC2* має окремий подільник частоти і можливість працювати асинхронно від зовнішнього «годинникового», із частотою 32 кГц кварцового резонатора. Для генерації сигналів кожен таймер-лічильник може керувати сигналом на двох цифрових лініях, які суміщені із лініями портів. Для таймера *TC0* лінії називаються *OC0A, OC0B* (суміщені із PD6 (лапка №10), PD5 (лапка №9), для таймера *TC1 – OC1A, OC1B* (суміщені із PB1 (лапка №13), PB2 (лапка №14), для таймера *TC2 – OC2A, OC2B* (суміщені із PB3 (лапка №15), PD3 (лапка №1).

Всі таймери мають спільні риси: мають в своєму складі подібні модулі (модулі порівняння, генератори сигналів), подібні режими роботи та налаштування, тому розберемо детально будову та програмування таймера *TC1.*

Основні риси таймера лічильника *TC1:*

- розрядність лічильного регістру 16 біт;
- два незалежних модулі порівняння;
- модуль захвату;
- режим очищення лічильного регістру при співпадінні значення лічильного регістру та регістру порівняння (автоперезавантаження);
- генератор сигналу широтно-імпульсної модуляції;
- генератор цифрового сигналу із змінною частотою;
- підрахунок зовнішніх подій;
- наявність 4 незалежних джерел переривань.

Таймер-лічильник складається із модуля двонаправленого (рахунок може відбуватися в прямому напрямі – інкремент лічильного регістру TCNT1, або в зворотному – декремент лічильного регістру TCNT1) програмованого 16-розрядного лічильника, модуля захвату зовнішніх подій, модуля порівняння та генератора сигналів.

Модуль захвату зовнішніх подій: якщо сигнал, який вказує на зовнішню подію прикладений до виводу IC1, то вміст TCNT1 зберігається в спеціальному регістрі ICR1. Значення цього

регістру може бути використане для визначення частоти, тривалості та заповнення сигналу. Модуль порівняння має два спеціальні регістри OCR1A та OCR1B вміст яких постійно порівнюється із вмістом TCNT1 і при співпадінні значень може бути згенерований запит на переривання, або сформований цифровий сигнал на лапках OC1A та OC1B.
Для взаємодії із таймером на програмному рівні наявна сукупність регістрів, які можна умовно розділити на 3 типи:

1. 16-ти розрядні регістри даних,
2. регістри налаштування4
3. регістри для налаштування та обробки переривань.

До першого типу відносяться: лічильний регістр TCNT1, в якому накопичується число, регістри порівняння OCR1A, OCR1B та регістр захвату ICR1. Регістри налаштування TCCR1A, TCCR1B, TCCR1C є 8 розрядні, комбінації бітів в цих регістрах задають швидкість роботи лічильника, джерело вхідних імпульсів та режими роботи додаткових модулів. Регістри для управління перериваннями це регістр масок TIMSK1 та регістр прапорців переривань TIFR1.

Види роботи таймера-лічильника. Вид роботи таймера-лічильника та тип сигналу на вихідній лапці OC1A, OC1B буде визначатися комбінацією бітів WGM[3..0] (Waveform Generation Mode) та бітами в регістрі TCCR1A COM1A[1..0], COM1B[1..0] які відповідають за вид роботи в режимі порівняння (Output Compare Mode). Вони визначають поведінку виводів OC1A, OC1B – встановлені в 1, скинуті в 0, чи змінений стан на протилежний, при співпадінні значень в лічильному регістрі TCNT1 та регістрах порівняння OCRA та OCRB.

Нормальний режим роботи.

За нормального режиму роботи значення в лічильному регістрі збільшується на одиницю (інкрементується) із приходом кожного імпульсу. Лічильник рахує до максимального значення, яке називають MAX=0xFFFF, а потім, після переповнення лічильника, починає рахувати від початкового значення, яке називають BOTTOM=0x0000. В момент встановлення значення BOTTOM, встановлюється прапорець переповнення лічильника TOV в регістрі TIFR1, це може бути використано як 17-й біт, але обробляти його потрібно програмно. Нове значення в TCNT1 може бути записане програмно в будь-який час.

Режим очищення таймера при співпадінні *(Clear Timer on Compare Match – CTC mode).* В цьому режимі регістри OCR1A або ICR1 використовуються для маніпуляції із роздільною здатністю лічильника: значення в цих регістрах (його називають TOP) постійно порівнюється із числом в лічильному регістрі TCNT1 і при співпадінні цих значень в TCNT1 записується 0x0000. Ці регістри будуть визначати максимальне значення лічильника і, відповідно, його роздільну здатність. Часова діаграма режиму CTC наведена на рис. 1.

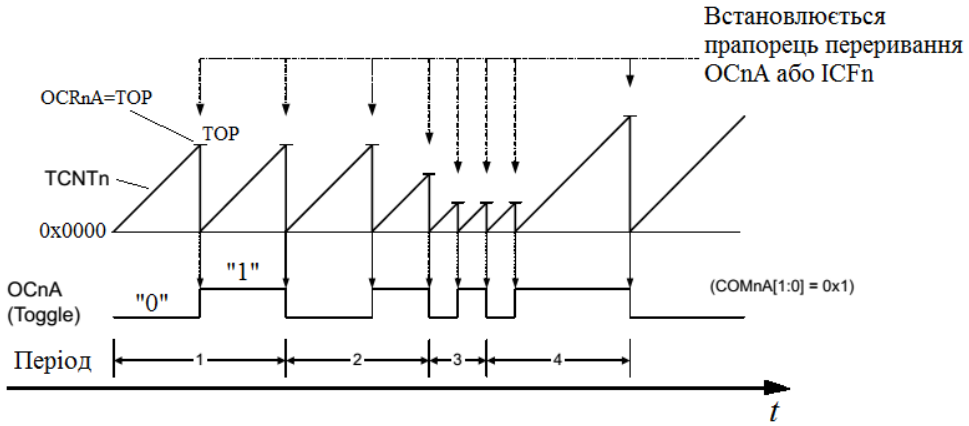


Рис. 1. Часова діаграму режиму CTC. OCRnA, TCNTn – значення регістру порівняння та лічильного регістру; OCnA – цифровий сигнал на виводі OCnA

Нехай, лічильник TC1 запрограмований на режим роботи «зміна стану виводу OC1A на протилежний при співпадінні вмісту регістрів OCR1A, TCNT1». З плином часу вміст регістру TCNT1 монотонно зростає і порівнюється із значенням TOP, а в момент часу коли TCNTn=OCRnA, відбуваються такі події: в регістрі TIFR1 встановлюється прапорець переривання OCRFA і створюється запит на переривання, в лічильний регістр TCNT1

записується значення BOTTOM, а сигнал на виводі OC1A змінюється на протилежний (Toggle). Лічильник продовжує рахувати з BOTTOM=0x0000 і кожного разу, коли TCNTn=OCRnA, буде знову генеруватися переривання та змінюватися на протилежній сигнал на OCnA. На цьому виводі буде генеруватися цифровий сигнал із частотою:

$$f_{OCnA} = \frac{F\_CPU}{2N(1+OCRnA)}, \quad (3.1)$$

де *F_CPU* тактова частота мікроконтролера, *N* коефіцієнт ділення частоти подільником (може приймати значення 1, 8, 64, 256, 1024), *OCRnA* - значення регістру порівняння. Відповідно до (3.1) за умови *N=1* та *OCRnA*=0 максимальна досяжна частота $f_{OCnA}^{MAX} = \frac{F\_CPU}{2}$.

Режим швидкого ШІМ (широтно-імпульсна модуляція). Широтно-імпульсна модуляція (Pulse-Width Modulation), або модуляція за тривалістю імпульсів, це спосіб отримати низькочастотний аналоговий сигнал довільного значення, змінюючи ширину цифрового сигналу. Приклад створення аналогового сигналу зображений на рис. 2. Ширина імпульсу модулюється аналоговим сигналом (тривалість імпульсу пропорційна до значення аналогового сигналу) (рис.2, а), а потім цифровий сигнал подається на інтегруючий пристрій (рис.2, б), і після інтегрування отримуємо аналоговий сигнал (рис. 2, в). Оскільки інтегруючий пристрій в частотній області працює як фільтр нижніх частот, то в частотній області цей процес можна описати як модуляцію високочастотного сигналу із подальшою фільтрацією в фільтрі нижніх частот.

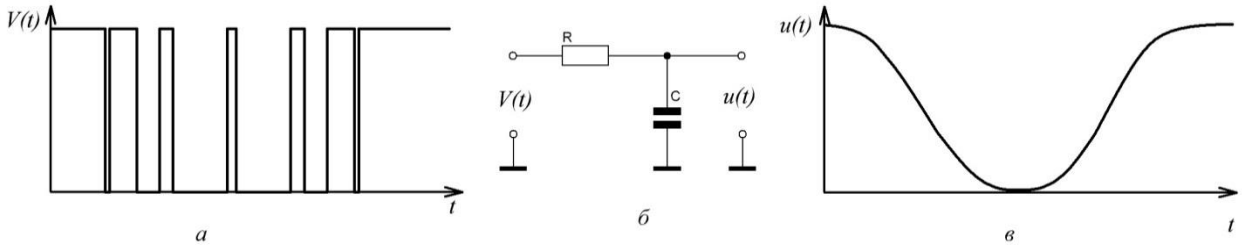


Рис. 2. Принцип отримання аналогового сигналу за допомогою ШІМ. В якості інтегруючого пристрою використана *RC* ланка.

Доволі часто, якщо результуючий аналоговий сигнал є не електричною величиною, в якості інтегратора виступає інерційність самої системи. Наприклад: при управлінні яскравістю свічення світлодіоду в якості інтегратора буде виступати око спостерігача, при управлінні швидкістю обертання електричного двигуна, інтегрування буде відбуватися за рахунок моменту інерції ротора.

Для створення сигналу ШІМ можна використовувати всі 3 наявні таймери *TC0, TC1* та *TC2*. Розрядність таймера-лічильника буде визначати точність відтворення аналогового сигналу, нехай діапазон зміни аналогового сигналу $u \ 0 \div 5\,B$, 8-ми розрядний таймер може генерувати сигнал тривалість якого можна розбити на $2^8 = 256$ частин. Відповідно, якщо аналоговий сигнал пропорційний до тривалості імпульсу, то дискретність відтворення аналогового сигналу буде рівна $\Delta u = \frac{5}{2^8} \approx 19{,}5\,мВ$. Якщо ШІМ буде 10-ти розрядний, то $\Delta u = \frac{5}{2^{10}} \approx 4{,}88\,мВ$. Для *TC0, TC2* розрядність ШІМ буде визначатися їх розрядністю: 8-біт, а таймер-лічильник *TC1* є 16-ти розрядний, і має можливість генерувати 8-, 9- 10-ти розрядний сигнал ШІМ, або довільної розрядності яка буде задаватися значенням TOP в регістрах OCR1A чи ICR1. Розрядність задається налаштуванням таймера за допомогою контрольних регістрів TCCR1A, TCCR1B.

Формування сигналу ШІМ відбувається наступним чином: в момент, коли значення регістру TCNTn (n –відповідає номеру таймера-лічильника) стає рівне BOTTOM, вихід OCnA встановлюється в «1». Коли TCNTn=OCRnA то вихід OCnA буде встановлений в «0» і генеруватися переривання по співпадінню. На відміну від режиму CTC, значення лічильного регістру не обнуляється, лічильник продовжує рахувати до переповнення. При досягненні максимального значення TCNTn=TOP (значення TOP буде задаватися роздільною здатністю

ШІМ: для 8-біт TOP= $2^8-1=255=0x00FF$, для 9-ти бітного $2^9-1=511=0x01FF$, для 10-ти бітного $2^{10}-1=1023=0x03FF$, а для 16-біт ШІМ $2^{16}-1=65535=0xFFFF$) генерується переривання по переповненню – встановлюється прапорець TOVn, регістр TCNTn стає рівний BOTTOM і OCnA встановлюється в «1». Такий цикл повторюється постійно і без участі процесорного ядра МК. Часова діаграма режиму швидкого ШІМ наведена на рис. 3.

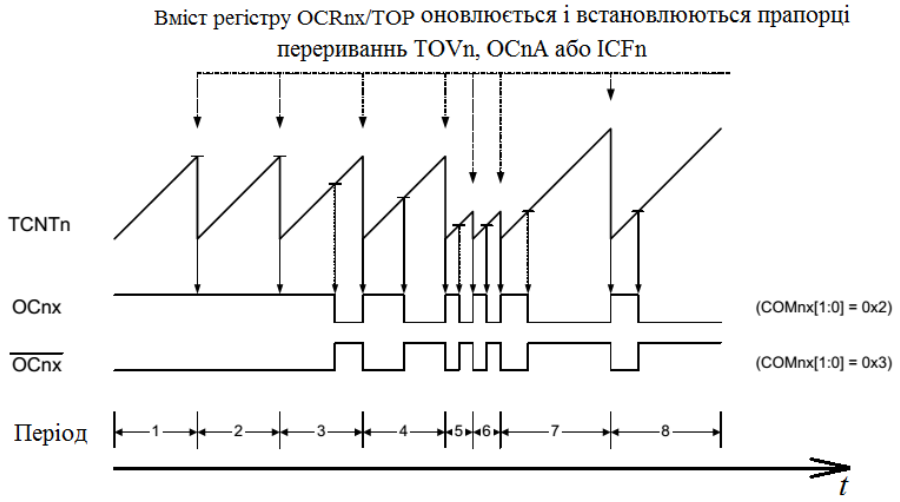


Рис. 3. Часова діаграму режиму швидкого ШІМ.

Із рис. 3 видно, що від значення TOP буде залежати частота сигналу ШІМ, а від значення в регістрі OCRnA буде залежати шпаруватість сигналу. Шпаруватість *(duty cycle)* це відношення часу включення до періоду виражене в відсотках, наприклад якщо є прямокутний сигнал де пів періоду «1», а пів періоду «0», то шпаруватість 50%).

Частота сигналу ШІМ може бути розрахована за формулою:

$$f_{OCnA}=\frac{F\_CPU}{N(1+TOP)}, \tag{3.2}$$

де *F_CPU* тактова частота мікроконтролера, *N* коефіцієнт ділення частоти подільником (може приймати значення 1, 8, 64, 256, 1024).

Граничними значенням для OCRnA буде BOTTOM (0x0000) – сигнал ШІМ буде дуже вузьким піком, якщо OCRnA=TOP то на виході буде постійний високий сигнал.

Режим фазокоректного ШІМ. Таймер-лічильник TC1 має режим фазокоректного ШІМ, який забезпечує більшу роздільну здатність та менші спотворення при генерації аналогових сигналів. Цей режим базується на використанні зворотної лічби в модулі лічильника. Лічильник рахує від BOTTOM до TOP, а потім від TOP до BOTTOM. Вихід OC1x буде скинутий в «0» при співпадінні значень TCNT1=OCR1A, при прямому рахунку (від BOTTOM до TOP) і встановлений в «1» в момент TCNT1=OCR1A, при зворотному рахунку (від TOP до BOTTOM). Оскільки, в цьому режимі використовується прямий та зворотній відлік, то частота сигналу ШІМ буде в 2 рази меншою, але точність ШІМ буде більшою. При зміні значення TOP сигнал буде змінюватися симетрично – і фаза сигналу не буде змінюватися. Такий режим ШІМ застосовується для управління двигунами. Часова діаграма режиму фазокоректного ШІМ наведена на рис. 4. Мінімальне значення TOP може бути 3 (відповідає 2-біт сигналу ШІМ), максимальне відповідає MAX. Частота сигналу фазокоректного ШІМ може бути розрахована за формулою:

$$f_{OCnA}=\frac{F\_CPU}{2\cdot N\cdot TOP}, \tag{3.3}$$

де *F_CPU* тактова частота мікроконтролера, *N* коефіцієнт ділення частоти подільником (може приймати значення 1, 8, 64, 256, 1024).

Режим ШІМ коректного за фазою та частотою. Такий режим подібний до попереднього - фазокоректного ШІМ, використовується як пряма, так і зворотня лічба в модулі лічильника. Але, якщо в процесі формування сигналу ШІМ потрібно змінювати його частоту, а це виконується шляхом зміни значення TOP в регістрі OCR1A або в ICR1, то оновлення цих регістрів відбувається тоді, коли TCNT1= BOTTOM. Це дозволяє завжди отримувати симетричний ШІМ сигнал. Частота сигналу ШІМ буде визначатися (3.3).

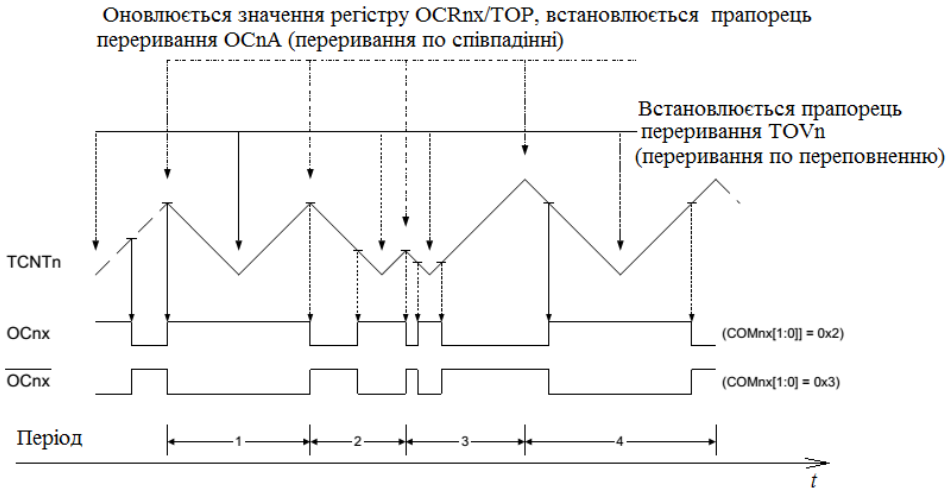


Рис. 4. Часова діаграма режиму фазокоректного ШІМ наведена на рис. 4.

Подільник частоти. Таймери-лічильники *TC0, TC1* мають спільний модуль подільника частоти. Подільник частоти являє собою 10-ти розрядний лічильник, на який подається системна тактова частота. Зменшення частоти за допомогою подільника дозволяє збільшити час який можуть відраховувати таймери-лічильники *TC0, TC1*. Ці таймери-лічильники можуть рахувати із системною частотою МК F_CPU – це найшвидший режим роботи. Або використовувати одну із 4-х частот після подільника частоти: $F\_CPU/8$, $F\_CPU/64$, $F\_CPU/256$ або $F\_CPU/1024$. Подільник частоти є незалежним від налаштувань *TC0, TC1* і формує всі ці частоти одночасно. Це означає, що при використанні подільника частоти час появи першого імпульсу на вході таймера-лічильника є невизначений і може коливатися в межах від 1 тактового імпульсу до *N+1* імпульсів, де *N* це значення подільника: 8, 64, 256 чи 1024. Якщо потрібно уникнути такої неоднозначності, потрібно синхронізувати подільник частоти та таймер-лічильник. Це можна виконати за допомогою регістру *GTCCR (General Timer/Counter Control Register)*:

bit	7	6	5	4	3	2	1	0
GTCCR	**TSM**	-	-	-	-	-	**PSRASY**	**PSRYNC**

Біт **PSRYNC** відповідає за скидання в 0 подільника частоти для таймерів-лічильників *TC0, TC1*. Якщо цей біт встановлений в 1, то в подільник частоти записується 0, а **PSRYNC** скидається автоматично.

Біт **PSRASY** відповідає за скидання в 0 подільника частоти для таймера-лічильника *TC2.* Якщо цей біт встановлений в 1 то в подільник частоти записується 0, після цього **PSRASY** скидається автоматично.

Біт **TSM** встановлює режим синхронізації. Якщо записати в нього 1 то таймери лічильники будуть зупинені і подільники частоти будуть скинуті. При записі 0 в **TSM** подільник частоти і таймери лічильники починають рахувати одночасно.

Регістри таймера-лічильника TC1.

Регістри даних

TCNT1 – 16-ти розрядний регістр, де зберігається поточне значення лічильника.

OCR1A та **OCR1B** - регістри порівняння, в які поміщають значення, які постійно порівнюються із значенням в регістрі TCNT1**.** При співпадінні цих значень може генеруватися переривання, або змінюватися стан ліній OC1A чи OC1B.

ICR1 – регістр захвату, в цей регістр поміщається значення **TCNT1** кожного разу, якщо приходить сигнал на лінію ICP1. Окрім того цей регістр може бути використаний для визначення TOP значення.

Контрольні регістри:

TCCR1A (Timer/Counter Control Register A)

bit	7	6	5	4	3	2	1	0
TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10

Біти COM1A[1..0], COM1B[1..0] контролюють поведінку виходів OC1A та OC1B. Якщо будь-який із цих бітів встановлений, то відповідні лінії введення-виведення портів (PB1, PB2) будуть підключені до модуля порівняння, але для коректної роботи цих ліній потрібно їх сконфігурувати на вихід: DDRB|=(1<< PB1)|(1<< PB2). Режими роботи виходів OC1A та OC1B в залежності від конфігурації бітів COM1A[1..0], COM1B[1..0] у випадку Нормального режиму чи режиму CTC описані в таблиці 1:

Таблиця 1. Поведінка OC1A/OC1B при співпадінні значень в TCTN1 та OCR1A/OCR1B

COM1A1/ COM1B1	COM1A0/ COM1B0	Опис
Нормальний режим, або Режим співпадіння		
0	0	Нормальний режим роботи порту, OC1A/OC1B відключені.
0	1	Переключення сигналу OC1A/OC1B на протилежний, при співпадінні значень в TCTN1 та OCR1A/OCR1B
1	0	Очищення (встановлення низького рівня сигналу) OC1A/OC1B при співпадінні
1	1	Встановлення (встановлення високого рівня сигналу) OC1A/OC1B при співпадінні
Режим швидкого ШІМ		
0	0	Нормальний режим роботи порту, OC1A/OC1B відключені.
0	1	WGM[3:0]=14 або 15 OC1A перемикається на протилежний, за умови співпадіння, OC1B відключений (нормальний режим роботи порту)
1	0	Очищення OC1A/OC1B при співпадіні, встановлення OC1A/OC1B при BOTTOM (не інвертуальний режим)
1	1	Встановлення OC1A/OC1B при співпадіні, очищення OC1A/OC1B при BOTTOM (інвертуальний режим)
Режим фазокоректного ШІМ		
0	0	Нормальний режим роботи порту, OC1A/OC1B відключені.
0	1	WGM1[3:0]=9 або 11: OC1A перемикається на протилежний, за умови співпадіння, OC1B відключений (нормальний режим роботи порту)
1	0	Очищення OC1A/OC1B при співпадіні при прямому підрахунку, встановлення OC1A/OC1B при співпадіні при зворотному підрахунку
1	1	Встановлення OC1A/OC1B при співпадіні за прямого підрахунку, очищення OC1A/OC1B при співпадіні за зворотного підрахунку

Біти WGM11, WGM10 задають режим роботи генератора сигналів, більш детально описано в таблиці 3.

TCCR1B (Timer/Counter Control Register B).

bit	7	6	5	4	3	2	1	0
TCCR1A	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10

Біти ICNC1, ICES1 – використовуються для управлінням режимом захвату значення TCTN1 за зовнішньою подією (Input Capture), біт ICNC1 вмикає фільтрацію вхідного сигналу від шуму, ICES1 задає подію, якщо ICES1=0, то захват буде відбуватися за заднім фронтом сигналу, якщо ICES1=1, то за переднім.
Біти WGM13, WGM12 задають режим роботи генератора сигналів, більш детально описано в таблиці 3.

Комбінація бітів CS12, CS11, CS10 задають джерело імпульсів, які рахує таймер-лічильник TC1 (таблиця 2).

Таблиця 2. Опис комбінації бітів, які визначають джерело імпульсів для таймера-лічильника TC1, , або

CS12	CS11	CS10	Опис
0	0	0	Вхідні імпульси відсутні – лічильник TC1 зупинений
0	0	1	$F\_CPU/1$ частота імпульсів рівна тактовій частоті МК
0	1	0	$F\_CPU/8$ сигнал надходить від подільника частоти
0	1	1	$F\_CPU/64$ сигнал надходить від подільника частоти
1	0	0	$F\_CPU/256$ сигнал надходить від подільника частоти
1	0	1	$F\_CPU/1024$ сигнал надходить від подільника частоти
1	1	0	Зовнішнє джерело вхідного сигналу з виводу T1. Активний є задній фронт імпульсу.
1	1	1	Зовнішнє джерело вхідного сигналу з виводу T1. Активний є передній фронт імпульсу.

Комбінації бітів WGM13, WGM12, WGM11, WGM10, які знаходяться в регістрах TCCR1A, TCCR1B контролюють види роботи таймера-лічильника (нормальний режим, CTC та три види ШІМ), режим роботи генератора сигналів, задають джерело максимального значення лічильника (TOP). Ці комбінації описані в таблиці 3.

Таблиця 3. Опис бітів які визначають режим генерації сигналів

№	WGM13	WGM12	WGM11	WGM10	Режим роботи таймера-лічильника	TOP	Оновлення OCR1x	TOV1 встановлюється
0	0	0	0	0	Нормальний	0xFFFF	Негайно	MAX
1	0	0	0	1	8-біт фазокоректний ШІМ	0x00FF	TOP	BOTTOM
2	0	0	1	0	9-біт фазокоректний ШІМ	0x01FF	TOP	BOTTOM
3	0	0	1	1	10-біт фазокоректний ШІМ	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Негайно	MAX
5	0	1	0	1	8-біт швидкий ШІМ	0x00FF	BOTTOM	TOP
6	0	1	1	0	9-біт швидкий ШІМ	0x01FF	BOTTOM	TOP
7	0	1	1	1	10-біт швидкий ШІМ	0x03FF	BOTTOM	TOP
8	1	0	0	0	ШІМ із коректною частотою та фазою	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	ШІМ із коректною частотою та фазою	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	ШІМ із коректною фазою	ICR1	TOP	BOTTOM
11	1	0	1	1	ШІМ із коректною фазою	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Негайно	MAX
13	1	1	0	1	---	---	---	---
14	1	1	1	0	швидкий ШІМ	ICR1	BOTTOM	TOP
15	1	1	1	1	швидкий ШІМ	OCR1A	BOTTOM	TOP

TCCR1C

bit	7	6	5	4	3	2	1	0
TCCR1C	FOC1A	FOC1B	-	-	-	-	-	-

Біти FOC1A, FOC1B впливають на роботу таймера лічильника лише в режимі CTC, якщо встановити FOC1A/FOC1B, то відбудеться подія співпадіння по порівнянню і вихід OC1A/OC1B змінить свій стан відповідно до налаштувань (таблиця 1)

Регістри відповідальні за переривання.

TIMSK1

bit	7	6	5	4	3	2	1	0
TCCR1C	-	-	ICIE	-	-	OCIEB	OCIEA	TOIE

Біт ICIE, якщо встановлений в 1 і переривання дозволені глобально (I=1), то переривання при захоплені значення таймера-лічильника TC1 дозволене.

Якщо встановлені біти OCIEA/ OCIEB і переривання дозволені глобально то будуть дозволені переривання при співпадінні значення регістра TCNT1 та OCR1A/OCR1B.

Біт TOIE управляє перериванням по переповненню таймера-лічильника TC1. Якщо TOIE=1, то при досягненні значення регістра TCNT1=MAX відбудеться переривання.

TIFR1

bit	7	6	5	4	3	2	1	0
TIFR1	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1

Прапорці переривань ICF1, OCF1B, OCF1A, TOV1 встановлюються, якщо відбулося відповідне переривання. Прапорці скидаються автоматично після виконання відповідної до кожного переривання функції обробки ISR(vector). Прапорці можна скинути програмно, записавши в них 1.

Розглянемо декілька прикладів застосування таймерів-лічильників. Перший приклад: це динамічна індикація числа на семисегментному світлодіодному дисплеї (ССД) на 4 знакомісця. В лабораторній роботі №1 було розглянуто роботу ССД в режимі статичної індикації, коли кожен сегмент дисплея, який використовується для відображення, включений постійно. Таке включення вимагає 8 окремих ліній для кожного знакомісця, що для 4 знакомісць складе 32 окремі ліній. ATmega328P має 23 лінії вводу-виводу, тому для роботи із дисплеєм використовується динамічна індикація: в кожне знакомісце включається по черзі, і якщо це робити достатньо швидко, то за рахунок інерційності ока, будемо спостерігати всі символи одночасно. Катоди світлодіодів, які складають знакомісце, з'єднують із відповідники катодами інших трьох знакомісць – всі світлодіоди паралельно з'єднані між собою (рис. 5). Переключення знакомісць відбувається за рахунок управління анодами. Таким чином, для виведення зображення потрібно 8 ліній для катодів та 4 лінії управління для анодів, всього 12 ліній. Аноди підключимо до PORTB: PB0, PB1, PB2, PB3. Катоди підключимо до PORTD.

Розглянемо просту програму для ілюстрації динамічної індикації. Підключимо 2 кнопки до PC0 та PC1. При натисканні 1-ї кнопки число, яке виведене ССД буде зменшуватися на 1, а при натисканні другої - збільшуватися на одиницю. Для організації динамічної індикації використаємо 8-ми розрядний таймер-лічильник TC0. Час протягом, якого буде світитися одне знакомісце, буде відраховуватися за допомогою режиму CTC – очищення лічильного регістру TCNT0 при співпадінні значень регістрів TCNT0 та OCR0A. В цей момент буде генеруватися запит на переривання OCMP0A, в функції, яка є реакцією на переривання будемо перемикати аноди. Часовий інтервал свічення 1 знакомісця виберемо 3,2 мс. Для забезпечення такого інтервалу, вибираємо подільник частоти 1024 і значення для регістру OCR0A знаходимо за формулою:

$$OCR0A = \tau \frac{N}{F\_CPU}, \qquad (3.4)$$

де τ - часовий інтервал, який потрібно відміряти за допомогою таймера, для наших значень F_CPU - тактова частота МК, N – значення подільника частоти. Для вибраних значень τ =3,2 мс, F_CPU =16 МГц, N=1024, значення регістру OCR0A=50.

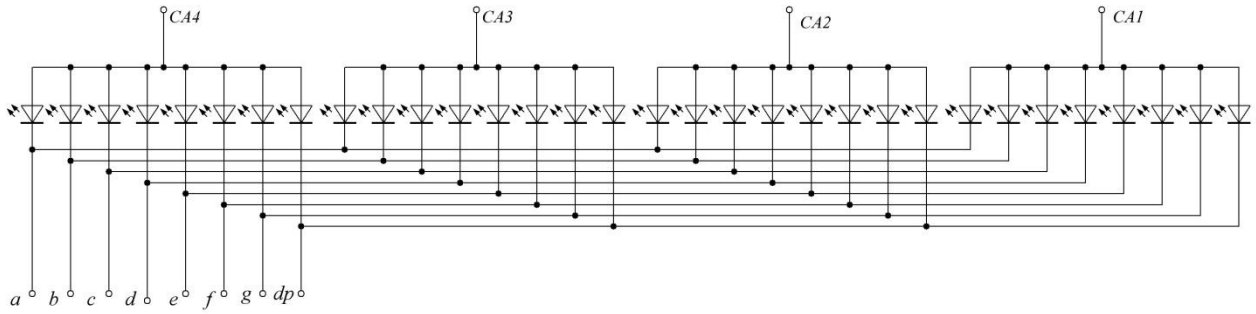


Рис. 5. Схема з’єднань семисегментного світлодіодного дисплея для динамічної індикації

Програмна реалізація динамічної індикації наведена в лістингу 3.1. Логіку роботи програми можна зрозуміти із коментарів.

Лістинг 3.1. Динамічна індикація за допомогою таймера-лічильника TC0.

```
/* Lab3-1.c
   Приклад програми із використанням переривань від таймера для організації
   динамічної індикації 7-ми сегментного світлодіодного дисплея (7LED).
   В програмі підраховується кількість натискань на кнопки при натисканні на 1-шу кнопку
   число збільшується на 1, а при натисканні на 2-гу кнопку зменшується на 1,
   отримане число виводиться на 7LED.
   Динамічна індикація - в даний момент часу світиться лише 1 розряд 7LED.
   Якщо перемикати розряди швидко - будемо бачити все число.
   Для того, щоб відміряти часові інтервали використовуємо таймер-лічильник TC0
   переривання по співрпадінню Output Compare A Match Interrupt
   значення лічильного регістра TCNT0 із значенням і регістрі порівняння OCR0A.
*/
/* ************************************************************
/        7-сегментний світллодіодний дисплей приєднаний до S7LED_Port:
/  S7LED_Port's bit     |7 |6|5|4|3|2|1|0|
/  7 seg LED            |dp|g|f|e|d|c|b|a|
/  7-сегментний світлодіодний дисплей із загальним анодом -
/  сегменти будуть світитися якщо рівень напруги - 0.
/ ************************************************************ */
// цю частину програми (до опису функцій) можна винести в окремий заголовний файл
#define F_CPU 16000000UL   // Частота роботи ATmega232P 16 МГц

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

#define LED_Button_PORT PORTC   // порт до якого буде приєднано  кнопка
#define LED_Button_DDR  DDRC      //  регістр вибору напрямку роботи порта кнопки
#define Button_PIN PINC         // регістр для перевірки стану кнопки
#define Kn_pin1 PC1            // кнопка приєднана до PC1
#define Kn_pin0 PC0            // кнопка приєднана до PC1

#define S7LED_Port PORTD    // 7-сегментний дисплей приєднаний до PORTD
#define S7LED_DDR DDRD      // регістр задає режим роботи (вхід-вихід) LED_7seg_Port

#define Anode_Port      PORTB //  аноди 7-ми сегментного індикатора приєднати до PORTB
#define Anode_Port_DD   DDRB

void Int_Port(void);      // початкова ініціалізація портів
unsigned char Chk_Kn(char k);       // перевірка стану кнопки
void Init_Timer0(void);           // ініціалізація таймера-лічильника TC0

/* функція яка перетворює 10-ве число Data в набір цифр: 1234 - "1", "2", "3", "4", які
поміщаються в масив Number_String */
void Int_To_Array( unsigned char* Number_String, int Data);

// Глобальні змінні
// кодування цифр в 7-сег. код:
```

```
unsigned char Digits[11]={0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8, 0x80, 0x90, 0xFF};
unsigned char Number_String[4]={0}; //масив для збереження числа у вигляді сукупності цифр

//лічильник в якому зберігається поточне значення номеру цифри числа, яке виводиться на 7LED
unsigned char i=0;
int n=0;             // змінна, в якій зберігається кількість натискань на кнопку

//---------- Опис функцій -------------------------
void Int_Port(void)
{
/* лінія кнопки - вхід, підключаємо підтяжучий резистор, якщо не підключити, то розрив в
 колі кнопки буде читатися як натиснута кнопка */
    LED_Button_DDR &= ~(1<<Kn_pin0);
    LED_Button_DDR &= ~(1<<Kn_pin1);
    LED_Button_PORT |= (1<<Kn_pin0);
    LED_Button_PORT |= (1<<Kn_pin1);

    S7LED_DDR = 0xFF;     // порт до якого підключений 7-ми сег. дисплей - вихід
    S7LED_Port = 0xFF;     // всі лінії порту  1 - дисплей вимкнений

    Anode_Port_DD = 0xFF;  //  порт до якого приєднані аноди 7LED - вихід
    Anode_Port = 0x00;    // на аноди подаємо 0 В
}

// фукція перевірки стану кнопки, повертає 1, якщо кнопка натиснута, 0 якщо ні
unsigned char Chk_Kn(char k)
{
   volatile char Kn=0;
   if ( ((~(Button_PIN)) & (1<<k)) )   // перевіряємо чи натиснута кнопка
       {
        _delay_ms(5);                    // боремося із "брязкотом" контактів  кнопки
        Kn=((~(Button_PIN))&(1<<k))>>k;   // ще раз перевіряємо стан кнопки
              while((~Button_PIN)&(1<<k))       // чекаємо на відпускання кнопки
              {
               _delay_ms(1);
              }
       }
       return Kn;
}
/*
перетвоюємо ціле число Data в упорядкований набір цифр, який зберігається в масиві
Number_String  у вигляді: кількість тисяч, кількість сотень, кількість десятків, кількість
одиниць. Ці цифри будуть використані, як індекси для виборки 7LED коду цифри із масиву
Digits
*/
void Int_To_Array( unsigned char* Number_String, int Data)
 {
   // зануляємо попереднє знаення
   Number_String[0] = Number_String[1] = Number_String[2] = Number_String[3] = 0;
   i = 0; // відображення цифри починаємо із початку
     while (Data>=1000)              // вичисляємо кількість тисяч
     { Data -= 1000; Number_String[3]++; }
     while (Data >= 100)             // вичисляємо кількість сотень
     { Data -= 100; Number_String[2]++; }
    while (Data> = 10)                        // вичисляємо кількість десятків
     { Data -= 10; Number_String[1]++; }
     Number_String[0] += Data;              // те що лишилося - кількість одиниць

 // якщо старші розряди 0, то їх не відображаємо
 if(Number_String[3]==0) Number_String[3]=10;
  {
   if (Number_String[3]==10 && Number_String[2]==0) {Number_String[2]=10;}
   if (Number_String[3]==10 && Number_String[2]==10 && Number_String[1]==0)
        { Number_String[1]=10;}
  }
}
/* Ініціалізація таймера TC0 - генерація запиту на переривання, при співпадінні вмісту
```

```
регістрів TCNT0 та OCR0A */
void Init_Timer0(void)
{
 TCCR0A |= (1<<WGM01); // вибираємо режим CTC
 TIMSK0 |=( 1<<OCIE0A);// Дозволяємо переривання співпадінні вмісту регістрів TCNT0 та OCR0A
 OCR0A = 50; // інтервал часу буде визначатися вміст OCR0A*1024/16 000 000 = 3,2 мс
 TIFR0 |= (1<<OCF0A); // скидаємо прапорець переривання
 TCNT0 = 0x00; //записуємо початкове значення в лічильний регістр

 //включаємо лічильник та встановлюємо подільник частоти F_CPU/1024
 TCCR0B |= (1<<CS02)|(1<<CS00);
}

ISR(TIMER0_COMPA_vect) //функція, яка є реакцією на переривання
{
    Anode_Port=0x00;  // аноди вимкнули щоб погасити 7LED
// в порт до якого приєднані катоди 7LED вивели код поточної цифри
    S7LED_Port=(Digits[Number_String[i]]);
    Anode_Port=(1<<i);  // включили відповідний анод
/* лічильник цифри збільшили на 1, якщо вийшли за межі кількості розрядів 7LED
   то починаємо із початку */
    if (++i>3) i=0;
}
int main(void)
{
    Int_Port();
    Init_Timer0();
    n = 1234; // виводимо перше число 1234, для перевірки правильності підключення анодів
    Int_To_Array(Number_String, n);
    sei(); // дозволяємо переривання
    _delay_ms(1000); // затримка в 1 с
    n = 0;
    Int_To_Array(Number_String, n);

while (1)
{
   if(Chk_Kn(Kn_pin1)) // перевіряємо стан кнопки 1
    {
     if (++n>9999) // перевіряємо чи не вийшли за верхню границю
      {
       n=9999;
      }
     Int_To_Array(Number_String,n); // перетворюємо число в масив
    }
   if(Chk_Kn(Kn_pin0)) // перевіряємо стан кнопки 0
    {
     if (--n < 0) // перевіряємо чи не вийшли за нижню границю
      {
       n=0;
      }
     Int_To_Array(Number_String,n);
    }
}
}
```

Для виведення зображення числа на ССД потрібно його розкласти на цифри: перетворити його із числового значення на сукупність цифр, які його зображають. Ця операція виконується в функції `void Int_To_Array( unsigned char* Number_String, int Data)`, в яку передаються аргументи: вказівник на масив із чотирьох елементів `Number_String`, де будуть зберігатися цифри та число `Data`. Початкове налаштування TC0 на потрібний режим роботи проводиться за допомогою функції `void Init_Timer0(void)` (детальний опис регістрів, які були використані див. технічний опис ATmega323P ст. 138, розділ 19.9 Timer/Counter0, Register Description).

Застосування широтно-імпульсної модуляції.

Якщо змішувати світло червоного, зеленого та синього кольорів в різних пропорціях, то можна отримати довільний колір. Технічна реалізація такого способу, отримання різних кольорів, можлива за допомогою RGB світлодіоду. RGB світлодіод – це пристрій в якому в одному корпусі об'єднано 3 світлодіоди, які світять червоним (R), зеленим (G) та синім (B) кольором. Для управління RGB світлодіоду потрібно 3 канали 8-ми бітного ШІМу – по одному каналу на колір. Яскравість свічення світлодіоду пропорційна до заповнення сигналу ШІМ: 0 – не світиться, 127 – 50%, 255 - 100%. Для отримання потрібного кольору потрібно знати пропорції змішування, їх можна взяти із довільного графічного редактора: наприклад встановлений на кожному ПК «Paint» - меню "Edit Colors" (рис. 6).

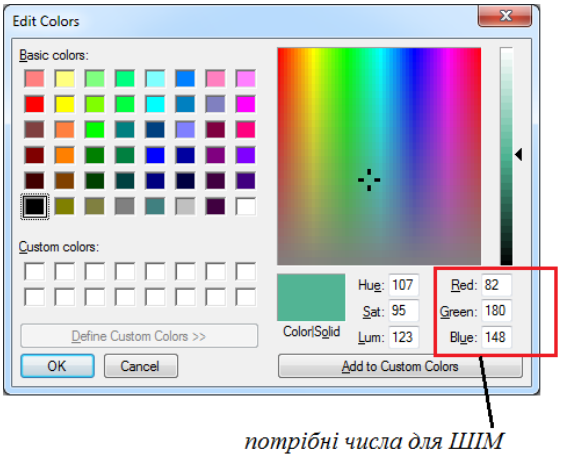


Рис.6. Пропорції змішування RGB кольорів для отримання довільного кольору.

Програмна реалізація управління кольором RGB світлодіоду за допомогою ШІМ модуляції наведена в лістингу 3.2. Логіку роботи програми можна зрозуміти із коментарів.

Лістинг 3.2. Управління кольором RGB світлодіоду за допомогою ШІМ модуляції

```
/*
PWM1.c
управління кольором RGB світлодіоду, за допомогою ШІМ модуляції. В залежності від
прогальності кожного сигналу, інтенсивність свічення кожного кольору буде різною кольори:
червоний, зелений, синій змішуючись, будуть давати довільний колір використовується режим
швидкого ШІМ в ТС0, ТС2
 */
#define F_CPU 16000000
#include <avr/io.h>
#include <util/delay.h>

void Init_Ports(void);
void Timer0_Init(void);
void Timer2_Init(void);

void Init_Ports(void)
{
/* Виводи PD5 (OC0B), PD6 (OC0A), PD3 (OC2B) - вихід формують ШІМ сигнали для кожного
кольору */
	DDRD |= (1<<PD6)|(1<<PD5)|(1<<PD3);
}
void Timer0_Init(void)
{
/* Налаштовуємо  ТС0 на режим швидкого ШІМ по двох каналах */
	TCCR0A |= (1<<COM0A1)|(1<<COM0B1)|(1<<WGM01)|(1<<WGM00);
	TCCR0B |= (1<<CS02); //включаємо ТС0, подільник частоти 256
/* записуємо максимальне значення - максимальна інтенсивність - біле свічення */
	OCR0A = 255;
	OCR0B = 255;
}
void Timer2_Init(void)
```

```
{// Налаштовуємо  TC2 на режим швидкого ШІМ по B каналу
       TCCR2A |= (1<<COM2B1)|(1<<WGM21)|(1<<WGM20);
       TCCR2B |= (1<<CS02); //включаємо TC2, подільник частоти 256
       OCR2B = 255;
       }

int main(void)
{
       Init_Ports();
       Timer0_Init();
       Timer2_Init();
       _delay_ms(2000);
       unsigned char i=0;

    while (1)
    {
       _delay_ms(50);
       OCR0B = i/2;   // R
       OCR0A=i;      // G
       OCR2B=255-i;  // B
       if (++i>254) { i = 0; }
    }
}
```

Генерація гармонічного сигналу за допомогою ШІМ. Як було описано вище, періодично змінюючи прогальність сигналу ШІМ (пропорційно до рівня аналогового сигналу в даний момент) можна генерувати довільний сигнал (рис.2.). Використаємо цей принцип для генерації гармонічного сигналу. Використаємо 9-ти бітний швидкий ШІМ TC1, розіб'ємо період на 128 частин і амплітуді в кожен момент часу поставимо у відповідність число, яке буде пропорційне до значення синуса:

$$N = 255,5\left[1+\sin\left(\frac{2\pi}{128}i\right)\right], i = 0..127\ , \qquad (3.5)$$

де N – число, яке пропорційне до амплітуди синуса і буде задавати ширину ШІМ сигналу, i – індекс який задає значення амплітуди в даний момент часу.

Для генерації використаємо два таймери-лічильники: TC1 буде генерувати ШІМ сигнал, а TC0 буде відміряти часові проміжки, за допомогою переривання і змінювати значення N в регістрі порівняння *OCR1A*. Оскільки ШІМ у 9-ти бітний, то максимальне значення $N=511$, ця умова накладає обмеження на мінімальний час зміни значення в *OCR1A* – мінімальний час, який повинен відрахувати TC0: $N = \frac{511}{16\times10^6} = 3,2\ \mu c$, це дає нам період гармонічного сигналу $T = 128\cdot 3,2\ \mu c \approx 4,1\ mc$, або частоту 244 Гц. Реалізація такого алгоритму наведена на лістингу 3.3.

Лістинг 3.3. Генерація гармонічного сигналу за допомогою ШІМ

```
/* lab3-3.c
 * Генерація sin(2*Pi*f*t) за допомогою швидкого 9-біт ШІМ
 */
#define F_CPU 16000000
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

// масив в якому зберігається значення пропорційні до амплітуди sin – визначатимуть //
заповнення сигналу ШІМ
const unsigned int Sin_table[128] =
         {256, 268, 281, 293, 305, 318, 330, 342, 353, 365, 376, 387, 397, 408, 418, 427,
          436, 445, 453, 461, 468, 475, 481, 486, 492, 496, 500, 503, 506, 508, 510, 511,
          511, 511, 510, 508, 506, 503, 500, 496, 492, 486, 481, 475, 468, 461, 453, 445,
          436, 427, 418, 408, 397, 387, 376, 365, 353, 342, 330, 318, 305, 293, 281, 268,
          256, 243, 230, 218, 206, 193, 181, 169, 158, 146, 135, 124, 114, 103, 93, 84, 75,
         66, 58, 50, 43, 36, 30, 25, 19, 15, 11, 8, 5, 3, 1, 0, 0, 0, 1, 3, 5, 8, 11, 30,
         36, 15, 19, 25, 43, 50, 58, 66, 75, 84, 93, 103, 114, 124, 135, 146, 158, 169,
```

```
          181, 193, 206, 218, 230, 243};

unsigned char i=0; // індекс для виборки поточного значення sin із масиву Sin_table

// Ініціалізація таймера TC0 - генерація запиту на переривання при співпадінні вмісту
// регістрів TCNT0 та OCR0A
void Init_Timer0(void)
{
 TCCR0A |= (1<<WGM01); // вибираємо режим CTC
 TIMSK0 |= (1<<OCIE0A);// Дозволяємо переривання співпадінні вмісту регістрів TCNT0 та OCR0A
 OCR0A = 8; // інтервал часу буде визначатися вміст OCR0A*64/16 000 000 = 3,2 мкс
 TIFR0 |= (1<<OCF0A); // скидаємо прапорець переривання
 TCNT0 = 0x00; //записуємо початкове значення в лічильний регістр
//включаємо лічильник та встановлюємо подільник частоти F_CPU/64
  TCCR0B |=(1<<CS01)|(1<<CS00);
}

ISR(TIMER0_COMPA_vect) //функція, яка є реакцією на переривання і відміряє відрізки часу
{
  OCR1A=Sin_table[i]; //записуємо в OCR1A наступне значення із Sin_table
   i += 1;
  if (i>127) { i = 0; }
}

void Init_Timer1(void)
{
       TCNT1 = 0;
       OCR1A = 0;
// задаємо режим роботи: 9-біт швидкий ШІМ
       TCCR1A |= (1<<COM1A1)|(1<<WGM11);
       TCCR1B |= (1<<CS10)|(1<<WGM12);
}

int main(void)
{
       Init_Timer0();
       Init_Timer1();
       DDRB |= (1<<PB1); //лапка OC1A (PB1) – вихід ШІМ генератора включається на вихід
       DDRD |= (1<<PD7); // підключаємо світлодіод для демонстрації паралельної роботи
      sei();
    while (1)
    { // оскільки ШІМ генерується апаратно та за допомогою переривань,
      //то МК може виконувати ще одну задачу – поморгати світлодіодом:
              PIND |= (1<<PD7);
              _delay_ms(250);
    }
}
```

Завдання до лабораторної роботи.

1. Створити проект в Atmel Studio 7, надрукувати програму наведену в лістинг 3.1. Вивчити логіку роботи програми за допомогою симулятора.
2. Створити проект в Atmel Studio 7, надрукувати програму наведену в лістинг 3.2. Вивчити логіку роботи програми за допомогою симулятора.
3. Створити проект в Atmel Studio 7, надрукувати програму наведену в лістинг 3.3. Вивчити логіку роботи програми за допомогою симулятора.

Розробити алгоритм, написати програму, відладити роботу за допомогою симулятора та перевірити за допомогою лабораторного стенду:

4. «Таймер зворотного відліку» - пристрій, який відміряє проміжки часу від 9999 до 1 сек. Відлік часу відбувається в зворотному напрямку, поточне значення відображається на ССД. Після того, як підрахунок дійде до 0 засвітити світлодіод, і створити звуковий сигнал частотою 1 кГц тривалістю 1 с. Для створення звукового сигналу використати

ШІМ модуляцію. Для управління використати 4 кнопки – «Старт», «Стоп», «Більше», «Менше». Кнопки «Більше», «Менше» встановлюють початкове значення інтервалу, а «Старт», «Стоп» - включення та зупинку таймера. Під час зворотного підрахунку «Більше», «Менше» - не активні. При утриманні кнопки «Стоп» більше секунди значення таймера обнуляється.

5. «Управління RGB світлодіодом за допомогою ШІМ» Розробити алгоритм та написати програму, яка за допомогою широтно-імпульсної модуляції регулює колір свічення RGB світлодіода. При включенні живлення світлодіод світиться білим кольором. Для управління використовуються 3 кнопки та семисегментний дисплей: одна кнопка відповідає за вибір кольору (R, G, B), а дві інші «Більше», «Менше». При натисканні кнопки 1 на дисплей виводиться в 1-й розряд назва кольору (Ч - червоний, З - зелений, С - синій), а інші 3 розряди виводиться число, яке задає шпаруватість ШІМ для цього кольору. При натисканні «Більше», «Менше» число може змінюватися межах 0-255. При відпусканні кнопки введене число задає шпаруватість ШІМ сигналу.
6. «Секундомір». Розробити алгоритм та написати програму, яка відміряє проміжки часу із точністю до сотих секунди в діапазоні від 99.99-0.00 секунд. При включенні живлення на 7-ми сегментному дисплеї висвічується 0.00 При натисканні кнопки «Старт» починається відлік часу із відображенням на 7-ми сегментному дисплеї. При повторному натисканні кнопки «Старт» відлік зупиняється. При натисканні кнопки «Скинути» на 7-ми сегментному дисплеї висвічується 0.00.
7. Генерація аналогових сигналів за допомогою ШІМ. Написати програму для генерації пилкоподібного та трикутного сигналу допомогою ШІМ модуляції.

Контрольні запитання:

1. Характеристики таймерів-лічильників ATmega328P. Із яких модулів складаються таймери-лічильники?
2. Види робочих режимів таймерів-лічильників.
3. Види переривань, які можуть генерувати таймери лічильники.
4. Особливості використання подільника частоти.
5. Широтно-імпульсна модуляція. Принцип дії, реалізація за допомогою таймера-лічильника.
6. Регістри даних, управляючі і контрольні регістри таймерів-лічильників TC0, TC2.
7. Регістри даних, управляючі і контрольні регістри таймера-лічильника TC1.

Література

1. Datasheet ATmega328P. http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf
2. Программирование микроконтроллеров ATMEL на языке С. Прокопенко В.С., К.: «МК-Пресс», 2012. – 320с.
3. Програмирование на языке С для AVR и PIC микроконтролеров. Шпак Ю.А., К.: «МК-Пресс», 2016. – 544стр.
4. Микроконтроллеры AVR семейств Tiny и Mega фирмы ATMEL. – М.: Издательский дом «Додэка-XXI», 2004. – 560 с.
5. Application Note_1497 AVR035: Efficient C Coding for 8-bit AVR microcontrollers. http://ww1.microchip.com/downloads/en/AppNotes/doc1497.pdf
6. Application Note_AVR42787 AVR Software User Guide http://ww1.microchip.com/downloads/en/AppNotes/Atmel-42787-AVR-Software-User-Guide_ApplicationNote_AVR42787.pdf

Лабораторна робота 4.

Використання вбудованого USART для взаємодії мікроконтролера та персонального комп'ютера.

Мета: навчитися використовувати вбудований в ATmega328P модуль універсального приймача-передавача USART.

Для забезпечення обміну даними МК із зовнішніми пристроями, часто використовується послідовна передача даних: якщо потрібно передати байт, то його біти передаються один за одним, по одній лінії зв'язку, таким чином, що в даний момент часу передається тільки один біт. Це дозволяє зменшити кількість ліній для передачі даних. Спеціальний апаратний модуль в складі МК ATmega328P, який забезпечує послідовну передачу даних, відповідно до стандарту послідовного інтерфейсу RS-232, називають універсальний синхронний/асинхронний послідовний приймач-передавач USART (Universal Asynchronous Receiver-Transmitter). Повна назва стандарту RS-232 «Інтерфейс між кінцевим обладнанням обробки інформації і кінцевим обладнанням каналу передачі даних, що використовує послідовний обмін двійковими даними» (Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange). Наприклад, персональний комп'ютер може мати «COM-port», який приймає та передає дані за стандартом RS-232, використовуючи його можна проводити обмін даними, пересилати команди між ПК та МК. Якщо у комп'ютера відсутній «COM-port», як у сучасних портативних комп'ютерах, то він може бути емульований за допомогою USB порту. Окрім ПК існує багато інших пристроїв, які використовують стандарт обміну даними RS-232: GSM-модеми, радіомодулі для бездротового зв'язку, GPS приймачі тощо.

Стандарт RS-232 передбачає передачу цілого байту по одній лінії у вигляді послідовності електричних імпульсів, кожен із яких може бути «1» чи «0». Обмін даними може бути синхронним – коли передача окремих бітів синхронізується за допомогою тактових синхроімпульсів, які передаються по додатковій лінії. При асинхронній передачі за допомогою спеціальних сигналів – стартові та стопові біти визначають початок та кінець передачі байту, і якщо перевіряти стан лінії у певні моменти часу після стартових бітів, то можна відновити байт, який був переданий.

Для асинхронного прийому та передачі потрібно дві сигнальні лінії, які називаються RxD (Receive Data) та TxD (Transmit Data) відповідно та лінія «землі» GND, відносно якої задаються рівні напруги логічного «0» та «1». Приклад з'єднання двох пристроїв наведений на рис. 1.

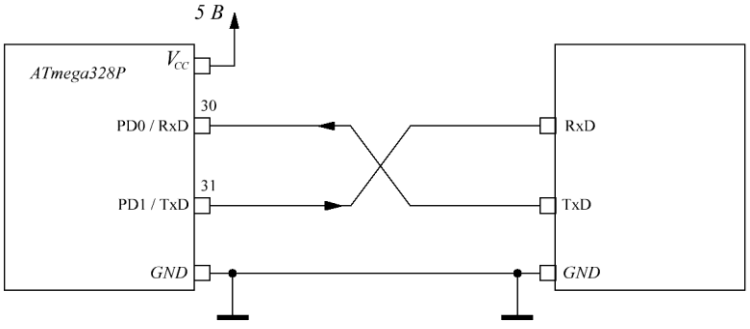


Рис. 1. З'єднання ATmega328P із зовнішнім пристроєм для обміну даними.

При асинхронному способі зв'язку потрібно, щоб приймач і передавач працювали із однаковою швидкістю. Число бітів, які передаються за 1 с називають бодрейт (baudrate), ці швидкості передачі є стандартизовані. Стандарт RS-232 встановлює логічні рівні: «1» - -3 ÷ -15 В, «0» +3 ÷ +15 В, в модулі USART логічні рівні відповідають TTL-рівням: «0» - 0 В, «1» - 5 В. Послідовність бітів, які передаються, називають ***формат кадру*** (frame format), кадр даних складається:

- 1 стартовий біт;
- 5, 6, 7, 8, 9 біт даних;
- наявний або відсутній біт парності;
- 1 або 2 стопових біти.

Якщо відсутня передача даних на лінії TxD, присутній високий рівень напруги – «1». При передачі даних кадр починається із стартового біту, за ним йдуть біти даних (від 5 до 9), першим завжди передається найменш значущий біт, а останнім найбільш значущий біт. Після даних можливий біт парності, за яким слідують 1 чи 2 стопові біти (рис. 2).

Рис. 2. Кадр даних: стартовий біт завжди «0», 5-9 біт даних, біт парності, 1 чи 2 стопових біти – завжди «1».

Оскільки біт парності та 2 стопових біти не є обов'язковими, то в більшості випадків кадр даних має вигляд: 1 стартовий, 8 біт даних, біт парності відсутній, 1 стоповий біт. Такий кадр називають 8n1 (8 data bits, no parity, 1 stop, bit).

Оскільки USART в ATmega328P оперує із логічними рівнями TTL, то для зв'язку персональним комп'ютером потрібно узгодити рівні і протоколи обміну. Якщо ПК має в своєму складі COM-port, логічні рівні якого відповідають стандарту RS-232 («1» - -3 ÷ -15 В, «0» +3 ÷ +15 В), то для електричного узгодження потрібно використовувати спеціальну мікросхему MAX232, або її аналог. У випадку відсутності COM-port, як в сучасних портативних ПК, для з'єднання МК та ПК використовують спеціальну мікросхему USB-USART міст, вона перетворює дані за стандартом RS-232 в дані стандарту USB. В ПК вона відображається як віртуальний COM-port. Для правильної роботи потрібно встановити драйвер мікросхеми, який доступний на сайті виробника. Як правило це мікросхема FT232 від *Future Technology Devices*: http://www.ftdichip.com/FTDrivers.htm,

або CH340G від *Jiangsu Qin Heng Co., Ltd*: http://www.wch.cn/download/CH341SER_ZIP.html.

Детальний опис USART, вбудованого в ATmega328P, наведений в технічному описі мікроконтролера в розділі 24, ст. 225. Налаштування робочих режимів USART відбувається за допомогою контрольних регістрів (детальний опис розділ 24.12 ст. 243 «Опис регістрів»):

UDR0	регістр даних – буфер пам'яті в який поміщається байт призначений для відправки та байт, який був отриманий
UCSR0A **UCSR0B** **UCSR0C**	контрольні та статусні регістри за допомогою яких задаються режими роботи USART та відповідних переривань
UBRR0L **UBRR0H**	регістри, які задають швидкість передачі та прийому даних

Початкова ініціалізація USART.

USART повинен бути налаштований перед першим використанням. Початкове налаштування включає: встановлення швидкості прийому-передачі (Baud Rate), встановлення формату кадру даних, включення приймача та передавача. Всі налаштування відбуваються шляхом встановлення відповідних бітів в регістрах **UCSR0A**, **UCSR0B, UCSR0C.** Якщо USART буде використовувати переривання, то під час налаштування потрібно глобально заборонити переривання.

Якщо потрібно провести зміну налаштування USART, то потрібно переконатися, що поточна передача чи прийом завершено. Приклад початкової ініціалізації USART наведено в лістингу 4.1.

Прийом, передача байту даних та використання переривань модуля USART.

Для здійснення прийому та передачі байту потрібно включити приймач та передавач, це виконується встановленням бітів TXEN та RXEN в UCSRB. Тоді виводи МК RxD, TxD (№ 30 PD0, 31 PD1) будуть підключені до USART. Для відправки байту даних потрібно переконатися чи завершена попередня передача та записати його в регістр даних UDR0 модуля USART. Прийом даних починається за наявності стартового біту кадру даних, потім кожен наступний біт буде зчитуватися із встановленою швидкістю і поміщатиметься у вхідний регістр зсуву. Після надходження стопового біту прийнятий байт із вхідного регістру зсуву переміщається у регістр даних **UDR0**, після цього він стає доступний до зчитування програмним способом.

При прийомі даних можна проводити перевірку прапорів помилок в регістрі **UCSR0A,** які повинні бути перевірені до зчитування регістру даних **UDR0**:

UPE – прапорець помилки контролю парності, який виставляється при виявлені помилки парності.

DOR – прапорець переповнення, який виставляється, при появі нового стартового біту за заповненого буферу приймача.

FE – прапорець помилки кадрування.

Для контролю процесу прийому та передачі в регістрі **UCSR0A** є спеціальні прапорці: RXC – прийнято новий байт даних, TXC – завершено передачу байту даних, UDRE – регістр даних **UDR0** порожній. Ці події можуть генерувати запити на переривання для обробки цих подій. Дозвіл на переривання визначатиметься встановленими в «1» відповідними прапорцями в регістрі **UCSR0B:**

RXCIE – дозвіл на переривання по звершенні прийому;

TXCIE– дозвіл на переривання по звершенні передачі;

UDRIE – дозвіл на переривання при опустошенні регістру UDR.

Приклади функцій, які реалізують прийом та передачу одного байту наведено в лістингу 4.2.

Опис регістрів.

UDR0 (USART In/Out Data Register). Буфер передавача та буфер приймача USART є фізично розділені, але вони мають одну адресу при програмному звернені до них. Якщо записати байт даних в **UDR0**, то він буде поміщений в буфер передавача, якщо читати дані із **UDR0** то буде надано доступ до буфера даних приймача.

UCSR0A (USART Control and Status Register 0 A).

bit	7	6	5	4	3	2	1	0
UCSR0A	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0

RXC0 (USART Receive Complete). Цей прапорець встановлюється, якщо є непрочитані дані в буфері приймача. Після зчитування даних він скидається.

TXC0 (USART Transmit Complete). Цей біт встановлюється, якщо закінчена передача байту даних із буфера передавача і відсутні нові дані в **UDR0.** Цей прапорець автоматично скидається, якщо було виконане переривання «закінчення передачі», або його можна скинути програмно записавши в нього «1».

UDRE0 (USART Data Registry Empty). Якщо цей прапорець встановлено, то **UDR0** є пустий і готовий прийняти новий байт.

FE0, DOR0, UPE0 – прапорці, які встановлюються у випадку помилок (помилка кадру даних, перезапис даних, помилка парності) при прийомі даних.

U2X0 – збільшує у два рази бодрейт USART.

MPCM0 – включає спеціальний режим роботи USART для між процесорного обміну даними.

UCSR0B (USART Control and Status Register 0 B).

bit	7	6	5	4	3	2	1	0
UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80

RXCIE0, TXCIE0, UDRIE0 – RXC0, TXC0, Complete Interrupt Enable, UDR0 Data Register Empty Interrupt Enable. Якщо ці прапорці встановлені, то переривання, які викликаються при

встановленні прапорців RXC0, TXC0, UDRE0 (USART_RX Complete, USART_TX Complete, USART_UDRE Data Register Empty), будуть локально дозволені.
RXEN0, TXEN0 (Receiver, Transmitter Enable). Встановлення цих бітів вмикає приймач та передавач в модулі USART.
UCSZ02 - разом із бітами **UCSZ0**[1..0] встановлює формат кадру.
RXB80, TXB80 – це 9-ті біти при прийомі та передачі, якщо встановлений формат кадру 9 біт даних.

UCSR0C (USART Control and Status Register 0 C).

bit	7	6	5	4	3	2	1	0
UCSR0C	UMSEL01	UMSEL00	UPM01	UPM00	USB0	UCSZ01	UCSZ00	UCPOL0

Біти цього регістру задають режим роботи USART (синхронний/асинхронний) встановлюють формат кадру даних, вмикають режим перевірки парності. При включені МК автоматично встановлюється асинхронний режим прийому-передачі, формат кадру 8n1 (8 біт даних, 1 стоп-біт, без перевірки парності). Такі налаштування використовуються в більшості випадків при обміну даними між ПК та мікроконтролерами.

UBRR0L, UBRR0H (USART Baud Rate Register Low, High). Сукупність цих двох регістрів є 12-бітний регістр **UBRR0**, вміст якого задає швидкість прийому-передачі в асинхронному режимі. Вміст **UBRR0** є значення подільника частоти, який формує робочу частоту USART. Формули для розрахунку бодрейту та значення регістру **UBRR0** наведені в таблиці 1.

Таблиця 1. Формули для розрахунку бодрейту та значення регістру **UBRR0.**

Режим роботи	Бодрейт	значення **UBRR0**
асинхронний, нормальний режим (U2X0=0)	$BAUD = \frac{F\_CPU}{16(UBRR0+1)}$	$UBRR0 = \frac{F\_CPU}{16BAUD} - 1$
асинхронний, подвійної швидкості (U2X0=1)	$BAUD = \frac{F\_CPU}{8(UBRR0+1)}$	$UBRR0 = \frac{F\_CPU}{8BAUD} - 1$

МК ATmega328P, який встановлений на лабораторному стенді має робочу частоту $F\_CPU = 16\ МГц$, приклади розрахунку значень **UBRR0** для цієї частоти наведено в таблиці 2.

Таблиця 2. Приклади розрахунку значення **UBRR0,** для стандартизованих швидкостей прийому передачі USART для робочої частоти МК $F\_CPU = 16\ МГц$.

Бодрейт, bps (bod per second)	асинхронний, нормальний режим (U2X0=0)		асинхронний, подвійної швидкості (U2X0=1)	
	UBRR0	Похибка	**UBRR0**	Похибка
2400	416	-0,1%	832	0%
4800	207	0,2%	416	-0,1%
9600	103	0,2%	207	0,2%
14,4 k	68	0,6%	138	-0,1%
19,2 k	51	0,2%	103	0,2%
28,8 k	34	-0,8%	68	0,6%
38,4 k	25	0,2%	51	0,2%
57,6 k	16	2,1%	34	-0,8%
76,8 k	12	0,2%	25	0,2%
115,2 k	8	-3,5%	16	2,1%
230,4%	3	8,5%	8	-3,5%
250 k	3	0%	7	0%
500 k	1	0%	3	0%
1 M	0	0%	1	0%

Похибка розраховується відносне відхилення розрахованої швидкості прийому-передачі від стандартизованої величини. За асинхронного режиму роботи похибка не повинна перевищувати 1,5%.

Приклади програм.

Початкова ініціалізація USART.

Лістинг 4.1. Функція початкової ініціалізації USART.

```
void USART_Init( unsigned int ubrr)
{
	// Встановлюємо швидкість прийому-передачі:
	//UCSR0A |= (1<<U2X0);
	UBRR0H = (unsigned char)(ubrr>>8);
	UBRR0L = (unsigned char)ubrr;
// Вмикаємо приймач, передавач, дозволяємо переривання по закінченню прийому байта даних:
	UCSR0B = (1<<RXEN0)|(1<<TXEN0)|(1<<RXCIE0);
}
```

Початкова ініціалізація зводиться до встановлення бодрейту (змінна `ubrr` її значення вибирається із таблиці 2, включення приймача і передавача, локального дозволу на переривання по завершенню прийому байта даних. При включенні формат кадру даних встановлюється 8n1. При потребі його можна змінити за допомогою конфігурації бітів **UCSZ0**[2..0].

Прийом та передача одного байту.

Лістинг 4.2. Функції передачі та прийому одного байту.

```
// Передача одного байту
void USART_Transmit( unsigned char data )
{
// Чекаємо поки закінчиться передача попереднього байту і звільниться буфер передавача
	while ( !( UCSR0A & (1<<UDRE0)) );
// Байт, який потрібно передати поміщаємо в буфер
	UDR0 = data;
}
// Прийом одного байту
unsigned char USART_Receive( void )
{
// Перевіряємо чи закінчений прийом
	while ( !(UCSR0A & (1<<RXC0)) );
// Отримуємо дані із буфера приймача і повертаємо значення
	return UDR0;
}
```

Якщо потрібно передати не один байт а символьний рядок то можна використати функцію наведену в лістингу 4.3.

Лістинг 4.3. Функція передачі символьного рядка.

```
unsigned char Hello_str[] = "Hello";// оголошуємо символьний рядок

void Srt_Transmit(unsigned char *str1) // передача символьного рядка
{
  unsigned char i = 0; // поточний індекс символа в рядку
  unsigned char Data; // змінна в яку поміщається символ із рядка

  while(1)
   {
     Data = str1[i++]; //вибираємо символ із рядка
       if (!Data) {// перевіряємо на рівність 0, якщо Data == 0 то дійшли до кінця рядка
         break; //якщо Data == 0, то дійшли до кінця рядка - виходимо із while(1)
       }
       while ( !( UCSR0A & (1<<UDRE0)) ); // чекємо закінчення передачі попереднього символа
       UDR0 = Data; // Data поміщаємо в буфер передавача
       ++i; // збільшуємо індекс на 1 - для вибірки наступного символу із рядка
   }
}
```

Символьний рядок це певна кількість комірок пам’яті, які розташовані підряд – одна за одною, як масив. На відміну від масиву, кількість символів в ній може бути довільною (не більше 256) і в кінці символьного рядку завжди поміщається 0. Тому при передачі символьного рядка не обов’язково знати кількість символів в рядку, достатньо передавати поки не досягнемо 0.

Для прийому та передачі даних від ПК до МК використовуються спеціальні термінальні програми. Таких програм є велика кількість у вільному доступі. Дуже зручною для використання є програма **Terminal 1.9 by Bray** вона є безкоштовною і не потребує інсталяції.

Функції на передачі і прийом одного байту даних та символьного рядку є простими, але мають суттєвий недолік: мікроконтролер витрачає час на очікування, на закінчення прийому, чи передачі байту і в цей час не може виконувати корисну задачу. Для уникнення такої ситуації можна передачу і прийом проводити використовуючи переривання по закінченню прийому та по очищенню буфера USART. Для організації обміну даними, із використанням переривань, потрібно виділити спеціальну область пам’яті – буфер, де будуть зберігатися дані, які призначені для передачі чи були прийняті через USART та розробити програмний механізм (набір функцій), який буде оперувати із даними та буфером (помістити байт в буфер, вибрати байт, передати чи прийняти байт). Зручним методом є *кільцеві буфери FIFO (First Input First Output).* Кільцевий буфер являє сукупність даних: масив `Buffer[BUFFER_SIZE]` розміром `BUFFER_SIZE`, та три змінні: `Index_Head` буде вказівником на перший байт який міститься в буфері, `Index_Tail` – це вказівник на останній байт в буфері, та `Counter` – змінна в якій міститься кількість байт в буфері (рис. 3). Розмір буферу вибирається більший за розмір даних, які можуть в нього записуватися за один раз. Запис наступної порції даних проводиться із «хвоста» буфера, і якщо доходимо до кінця буфера то продовжуємо запис із початку.

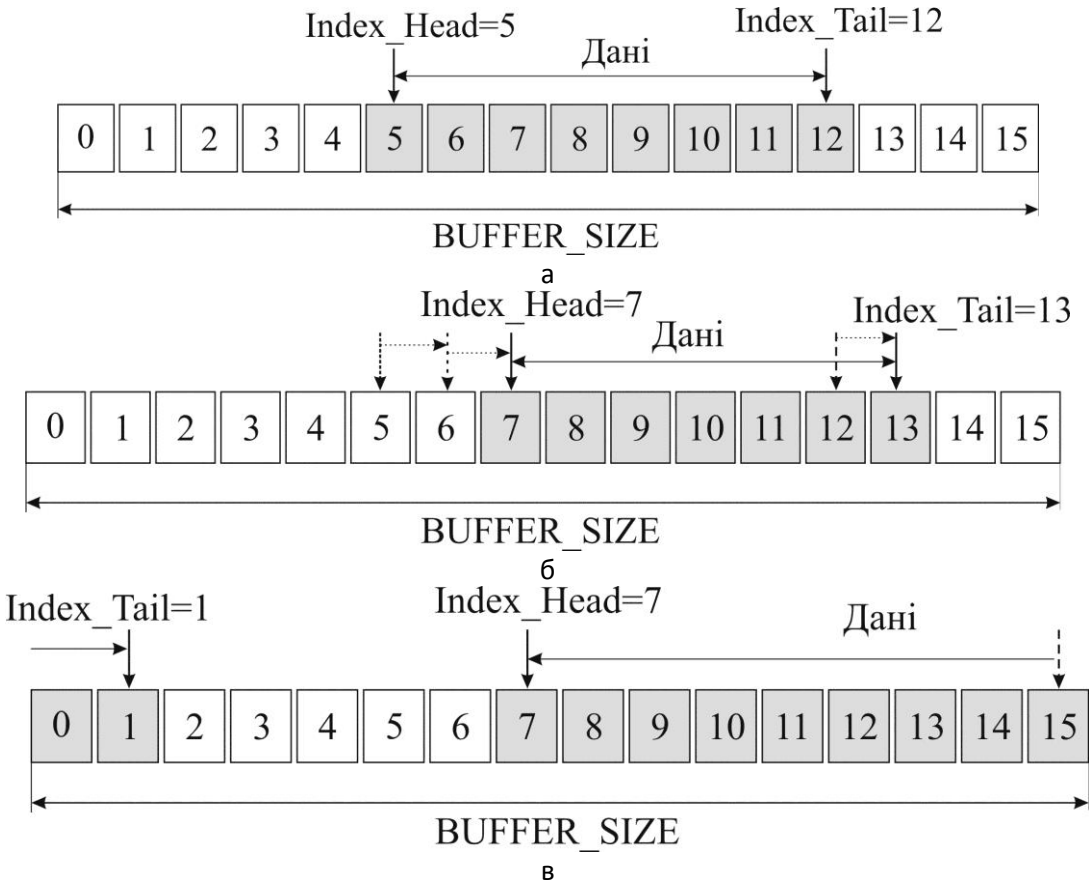


Рис. 3. Принцип роботи кільцевого буферу на 16 байт: а) розташування даних; б) переміщення індексів Index_Head та Index_Tail, при записі 1 байту та зчитуванні 2 байт; в) розташування Index_Tail після запису 4-х байт в кільцевий буфер.

При зчитуванні і запису даних в кільцевий буфер будуть змінюватися лише Index_Head, Index_Tail та Counter, а дані переміщатися не будуть. Для оптимізації роботи програми та зручності програмування, об'єднаємо ці змінні в структуру та створимо шаблон користувацького типу за допомогою декларації typedef:

```
typedef struct {
      unsigned char Buffer[BUFFER_SIZE];
      unsigned char Index_Tail;
      unsigned char Index_Head;
      unsigned char Counter;
      } Ring_Buffer;
```

Тепер, для створення кільцевого буфера, можна просто оголосити змінну типу Ring_Buffer.

Розглянемо принцип роботи кільцевого буфера. Якщо буфер порожній, то обидва індекси Index_Head та Index_Tail мають однакове значення (не обов'язково 0). Якщо потрібно записати байт Byte в буфер то він записується в «хвіст» буфера - місце куди вказує Index_Tail:

```
Buffer[Index_Tail] = Byte;
Index_Tail++;
Index_Tail &= BUFFER_MASK;
Counter++;
```

де BUFFER_MASK = BUFFER_SIZE-1. Index_Tail збільшується на одиницю він вказує на наступну комірку пам'яті в буфері, і збільшується на одиницю лічильник байтів Counter. Якщо Index_Tail вказує на кінець буфера, то потрібно його прирівняти до 0, щоб він вказував на початок буфера. Це можна проводити за допомогою оператора:
if (Index_Tail == BUFFER_MASK) { Index_Tail=0;}.
Якщо BUFFER_SIZE рівний 2^n, то цю операцію можна проводити за допомогою операції порозрядного І. Нехай BUFFER_SIZE = 16, тоді Index_Tail, Index_Head можуть приймати значення від 0 (0b00000000) до 15 (0b00001111), якщо Index_Tail чи Index_Head виходить за межі буфера то стає рівне 16 (0b00010000). Якщо виконати операцію порозрядного І із BUFFER_SIZE-1: 16 & 15 = 0 (0b00010000 & 0b00001111 = 0b00000000), то індекс стає рівним нулю автоматично. Якщо індекс є меншим за BUFFER_SIZE-1 то операція І його не змінює. Операція порозрядного І виконується набагато швидше за оператор if().
Зчитуються дані із кільцевого буферу з «голови», при цьому Index_Head збільшується на 1 буде вказувати на наступний елемент масиву Buffer, а Counter зменшуємо на 1:

```
Byte = Buffer[Index_Head];
Index_Head++;
Index_Head &= BUFFER_MASK;
Counter--;
```

При цьому потрібно слідкувати, щоб «голова» не перегнала «хвіст», тобто, якщо Index_Head == Index_Tail, чи Counter == 0, то непрочитаних даних немає і операцію зчитування припиняємо.

Якщо потрібно очистити кільцевий буфер, потрібно лише прирівняти до нуля змінні Counter, Index_Head та Index_Tail, а нові дані будуть записані поверх старих.
Для роботи із кільцевим буфером потрібно як мінімум 3 функції – функція очистки буфера, функція запису байту в буфер та функція зчитування байту із буфера.
Використання кільцевих буферів розглянемо на прикладі програми управляння станом світлодіодів через USART. ПК відсилає команди, які будуть накопичуватися в вхідному кільцевому буфері RxBuffer. Отримані команди будуть вибиратися із буфера та виконуватися. Після виконання команд буде надсилатися на ПК звіт через вихідний буфер TxBuffer. Для зручності подальшого використання програма розбита на декілька файлів: usart_ring_buffer.h, usart_ring_buffer.c (лістинги 4.4, 4.5) – файли, де описані

дані та функції потрібні для роботи із USART та кільцевими буферами. Головна програма `main.c` (лістинг 4.6), в якій обробляються та виконуються команди.
У заголовному файлі описані прототипи даних – користувацький тип `Ring_Buffer`, функції необхідні для роботи із USART:

```
void USART_Init(unsigned int ubrr); // початкова ініціалізація USART
void USART_Send_Byte(unsigned char Byte); // пересилка символа
void USART_Send_Str(unsigned char *Str); // пересилка символьного рядка
```

Функції для обслуговування кільцевого буфера:

```
// функція яка поміщає символ в кільцевий буфер
void PutByteBuffer(Ring_Buffer *pB, unsigned char Byte);
unsigned char GetByteBuffer(Ring_Buffer *pB); // функція, яка забирає символ із буфера
void ClearBuffer(Ring_Buffer *pB); // очистка буфера
```

Для доступу функцій до кільцевого буфера використовується вказівник `Ring_Buffer *pB.`

Лістинг 4.4. Заголовний файл `usart_ring_buffer.h`

```
/* Використання кільцевого буфера для обміну даними із ПК
   описані прототипи функцій та даних, які будуть використовуватися */
#define F_CPU 16000000UL

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

#define BUFFER_SIZE 64 //розмір буфера вибираємо кратний ступені 2^6
#define BUFFER_MASK BUFFER_SIZE-1 // маска для індексів буфера

/* всі змінні, які потрібні для роботи із кільцевим буфером, об'єднаємо в структуру: */
typedef struct {
       unsigned char Buffer[BUFFER_SIZE]; // буфер – масив для даних
/* змінна, де буде зберігатися індекс останнього записаного байта в
  масив Buffer[i] – «хвіст» буфера */
       unsigned char Index_Tail;
/* змінна, де буде зберігатися індекс першого записаного байта в
   масив Buffer[i] – «голова» буфера */
       unsigned char Index_Head;
       unsigned char Counter; // лічильник кількості байтів наявних масиві Buffer[i]
       } Ring_Buffer;

/* -------- прототипи функцій ----------------------------------------------*/
/*  USART    */
void USART_Init(unsigned int ubrr); // початкова ініціалізація USART
void USART_Send_Byte(unsigned char Byte); // пересилка символа
void USART_Send_Str(unsigned char *Str); // пересилка символьного рядка

/* функції для роботи із кільцевим буфером */
/* функція яка поміщає символ в кільцевий буфер */
void PutByteBuffer(Ring_Buffer *pB, unsigned char Byte);
unsigned char GetByteBuffer(Ring_Buffer *pB); // функція, яка забирає символ із буфера
void ClearBuffer(Ring_Buffer *pB); // очистка буфера
```

В файлі `usart_ring_buffer.c` описані зміні та функції, необхідні для обслуговування процесу обміну даних із кільцевими буферами `RxBuffer, TxBuffer.` Взаємодія із буферами відбувається за допомогою вказівників `pRxBuffer, pTxBuffer.` Передача та прийом даних через USART відбувається в перериваннях. Якщо був прийнятий байт, то в функції `ISR(USART_RX_vect)` він поміщається у вхідний буфер `RxBuffer` за допомогою функції `PutByteBuffer(pRxBuffer, Byte)`. Передача даних відбувається у 2 етапи: спершу за допомогою функції `USART_Send_Str(unsigned char *Str)` символьний рядок, який потрібно передати поміщається у вихідний буфер `TxBuffer` і дозволяється переривання по спустошенню UDR0, а потім, в функції `ISR(USART_UDRE_vect),` байт за байтом виймається із вихідного буфера і передається через USART.

Лістинг 4.5. Файл `usart_ring_buffer.c`

```
#include "usart_ring_buffer.h"

// кільцеві буфери для вхідних та вихідних даних
 Ring_Buffer RxBuffer;
 Ring_Buffer TxBuffer;

// вказівники на кільцеві буфери
 Ring_Buffer *pRxBuffer=&RxBuffer;
 Ring_Buffer *pTxBuffer=&TxBuffer;

// функція ініціалізації USART
void USART_Init(unsigned int ubrr)
{
// Встановлюємо швидкість прийому-передачі:
       UBRR0H = (unsigned char)(ubrr>>8);
       UBRR0L = (unsigned char)ubrr;
// Вмикаємо приймач, передавач, дозволяємо переривання по закінченню прийому байту даних:
       UCSR0B = (1<<RXEN0)|(1<<TXEN0)|(1<<RXCIE0);
}

void USART_Send_Byte(unsigned char Byte) //передача символу
 {
       while ( !( UCSR0A & (1<<UDRE0)) ); //перевіряємо чи вільний регістр даних UDR0
       UDR0 = Byte; // байт поміщаємо в UDR0
}
// функція поміщає символ у вихідний буфер
void PutByteBuffer(Ring_Buffer *pB, unsigned char Byte)
{
/* якщо кількість символів в буфері менше за розмір буфера то: */
 if (pB->Counter < BUFFER_SIZE)
  {
   pB->Buffer[pB->Index_Tail++] = Byte;  // записуємо в "хвіст" буфера
   pB->Index_Tail &= BUFFER_MASK;  // накладаємо маску для переходу на початок буфера,
   pB->Counter++;
  }
}
/* функція повертає символ із вхідного буферу */
unsigned char GetByteBuffer(Ring_Buffer *pB)
{
 unsigned char Byte = 0;
 if (pB->Counter>0) // перевіряємо чи є в буфері дані
  {
   Byte = pB->Buffer[pB->Index_Head]; // читаємо байт із "голови" - з початку запису
/*збільшуємо індекс на 1, оскільки байт прочитаний, то початком запису буде наступний байт*/
   pB->Index_Head++;
   pB->Index_Head &= BUFFER_MASK; // накладаємо маску для переходу на початок буфера
   pB->Counter--; // кількість непрочитаних байт в буфері зменшилася на 1
  }
 return Byte;
}

//  очищення буфера від даних

void ClearBuffer(Ring_Buffer *pB)
{ /* очищення буфера відбувається шляхом "забування", що буфері є дані
     індекси початку та кінця даних = 0, кількість байт = 0
     наступний запис в буфер почнеться з початку і старі дані будуть втрачені */
  pB->Index_Head = 0;
  pB->Index_Tail = 0;
  pB->Counter = 0;
}

void USART_Send_Str(unsigned char *Str)
{
/* символьний рядок спеціальний тип даних, описаний в <string.h>, який являє собою
   сукупність комірок пам"яті прозташованих підряд, як в масиві, але кількість комірок
```

```
   наперед не задано, в кінці рядка завжди поміщається 0x00 */
       unsigned char i = 0; // поточний індекс символа в рядку
       unsigned char Data; // змінна в яку поміщається символ із рядка
/* переміщаємо символьний рядок в вихідний буфер */
  while(1)
   {
     Data=Str[i++]; //вибираємо символ із рядка
     if (!Data) // перевіряємо на рівність 0, якщо Data == 0 то дійшли до кінця рядка
       {
         break; //якщо Data == 0 то дійшли до кінця рядка - виходимо із while(1)
       }
     PutByteBuffer(pTxBuffer, Data); // символ переміщаємо в буфер
   }
 /*Починаємо відправляти те, що перемістили в вихідний буфер -
   дозволяємо переривання по спустошенню UDR0, оскільки UDR0 від початку пустий, то це
   переривання буде активоване негайно після дозволу переривань за допомогою sei();*/
       UCSR0B |= (1<<UDRIE0);
}

ISR(USART_RX_vect) // переривання по закінченню прийому байту
 {
       unsigned char byte = UDR0; // читаємо отриманий байт
       PutByteBuffer(pRxBuffer, byte); // поміщаємо його в RxBuffer - буфер прийому
 }

 ISR(USART_UDRE_vect) // переривання по очищенню буфера передавача
{
 /* завантажуємо в UDR0 байт із буфера вихідного буфера TxBuffer
   UDR0 = GetByteBuffer(pTxBuffer);
 /* перевіряємо чи закінчена передача символів із pTxBuffer, якщо "голова" догнала хвіст,
   то передано було все наявне в буфері, зупиняємо передачу - забороняємо переривання
   по спустошенню UDR0 локально: */
       if (pTxBuffer->Index_Head == pTxBuffer->Index_Tail) {
          UCSR0B &= ~(1<<UDRIE0); // забороняємо переривання
        }
 }
```

В основній частині програми `main.c` кільцеві буфери використовуються для обміну командами та даними між ПК та МК. Від ПК приходитимуть команди управління світлодіодами (увімкнути/вимкнути), МК виконуватиме команди і надсилатиме звіт про виконання. Окрім того буде виконуватися фонова задача – «моргаючий» світлодіод. Для спрощення приймемо формат команд фіксованої довжини 4 символи:

Команда	Дія	Відповідь МК
cR1*	увімкнути червоний світлодіод	Red is On
cR0*	вимкнути червоний світлодіод	Red is Off
cG1*	увімкнути зелений світлодіод	Green is On
cG0*	вимкнути зелений світлодіод	Green is Off
cB1*	увімкнути синій світлодіод	Blue is On
cB0*	вимкнути синій світлодіод	Blue is Off
cS?*	надіслати звіт	LED Status: Red LED is Off Green LED is Off Blue LED is Off

Функції налаштування портів для світлодіодів є аналогічними до описаних в лабораторній роботі №1. Кільцеві буфери `RxBuffer, TxBuffer` та вказівники на них `pRxBuffer, pTxBuffer` оголошуються із специфікатором `extern`, оскільки вони описані в файлі `usart_ring_buffer.c`. Оскільки, прийняті дані від ПК поміщаються в вхідний буфер автоматично, то для виконання команд потрібно знайти в буфері `RxBuffer` перший символ команди, а потім перемістити 2 наступні символи в масив `Command[2]`, після цього встановлюємо прапорець `New_Comm_F=1` – сигнал того, що наявна нова команда до виконання. Цю операцію виконує функція `Get_Commad(pRxBuffer, Command)`, в яку

передаються вказівники на вхідний буфер `pRxBuffer` та на масив `Command`. Виконання отриманих команд відбувається в функції `Command_Execute(Command)`. Текст програми із детальними коментарями наведений в лістингу 4.5.

Лістинг 4.5. Файл `main.c`

```
/*
 *  lab4-1 Ring Buffer.c
 *  Приклад використання кільцевих буферів для обміну даними між ПК та МК
 *  ПК відправляє команди, які складаються із сукупності ASCII символів,
 *  отримані команди розшифровуються, виконуються і надсилається звіт про виконання
 *  До порту C під'єднані 3 світлодіоди Червоний, Синій, Зелений за командами від ПК будемо
     їх вмикати та вимикати
    Всі команди мають фіксовану довжину і починаються із символу c, а закінчуються *
       Наприклад:
          cR1* - увімкнути червоний світлодіод
          cR0* - вимкнути червоний світлодіод,
  аналогічні команди: cG1*, cG0*, cB1* cB0* - для зеленого та синього світлодіоду,
  після виконання команди надсилається звіт про виконання типу: Red is On
         cS?* - передати стан світлодіодів:
       Повертається повідомлення типу:
         Red is On
         Green is Off
         Blue is On
 */
#define F_CPU 16000000UL
#define BAUDRATE 51 // 19,2 kbps

// Світлодіод для фонової задачі
#define W_Led PD2
#define W_Led_DDR  DDRD
#define W_Led_PORT PORTD
#define W_Led_PIN  PIND

// Світлодіоди для основної задачі
#define rgb_DDR  DDRC
#define rgb_PORT PORTC
#define rgb_PIN  PINC

#define Red_Led PC0
#define Green_Led PC1
#define Blue_Led PC2

#include <avr/io.h>
#include <util/delay.h>
#include "usart_ring_buffer.h"

// Відповіді МК після виконання команди
unsigned char Ready[]="Ready\n"; // після включення сигнал готовності
unsigned char Status[]="LED Status:\n"; // початок звіту
unsigned char WTF[]="WTF???\n"; // відповідь на незрозумілу команду
unsigned char Red[]="Red LED is ";
unsigned char Green[]="Green LED is ";
unsigned char Blue[]="Blue LED is ";
unsigned char On[]="On\n";
unsigned char Off[]="Off\n";

// масив, де буде зберігатися поточна команда до виконання
unsigned char Command[2]={'0','0'};
// прапорець, який вказує на наявність нової команди: якщо команда була виконана = 0
unsigned char New_Comm_F=0;

/* оголошуємо кільцеві буфери, як extern - це означає, що ці структури були оголошені в
іншому файлі (usart_ring_buffer.c) */
extern Ring_Buffer RxBuffer;
extern Ring_Buffer TxBuffer;
// вказівники на кільцеві буфери
```

```
extern Ring_Buffer *pRxBuffer;
extern Ring_Buffer *pTxBuffer;

//-------  прототипи функцій -----------------------------------
void Blink(void); // фонова задача моргати світлодіодом
void Port_Init(char); // налаштування порту для фонової задачі
void RGB_Init(void);  // налаштування порту для RGB світлодіоду
void RGB_On(unsigned char Led); // вмикаємо світлодіод
void RGB_Off(unsigned char Led); // вимикаємо світлодіод
void Get_Status(void);  // перевіряємо стан світлодіодів
/* функція, яка вибирає команду із вхідного буфера */
void Get_Commad(Ring_Buffer *pB, unsigned char *Com);
void Command_Execute(unsigned char *Com); // виконання команди

//----------------------- опис функцій -----------------------------------
void Blink(void) // фонова задача, яка буде виконуватися постійно
{
  W_Led_PIN |= (1<<W_Led);
  _delay_ms(150);
}

void Port_Init(char L)
{
  W_Led_DDR |= (1<<L);
  W_Led_PORT |= (1<<L);
}

void RGB_Init() // ініціалізація порта для RGB світлодіодів
{
// лінії портів для світлодіодів на вихід
  rgb_DDR |= (1<<Red_Led)|(1<<Green_Led)|(1<<Blue_Led);
  rgb_PORT &= ~( (1<<Red_Led)|(1<<Green_Led)|(1<<Blue_Led) ); // світлодіоди вимкнули
}

void RGB_On(unsigned char Led) // функція, яка вмикає світлодіод
{
       rgb_PORT |= (1<<Led);
}

void RGB_Off(unsigned char Led) // функція, яка вимикає світлодіод
{
       rgb_PORT &= ~(1<<Led);
}

void Get_Status(void)
{
      // прочитали стан світлодіодів:
       unsigned char L = rgb_PORT & ( (1<<Red_Led)|(1<<Green_Led)|(1<<Blue_Led) );
       USART_Send_Byte('\n');
       USART_Send_Str(Status);
       // перевіряємо стан червоного світлодіоду
       if ( (L & (1<<Red_Led)) )   {USART_Send_Str(Red); USART_Send_Str(On);}
         else   {USART_Send_Str(Red); USART_Send_Str(Off);}
       // перевіряємо стан зеленого світлодіоду
       if ( (L & (1<<Green_Led)) )   {USART_Send_Str(Green); USART_Send_Str(On);}
       else   {USART_Send_Str(Green); USART_Send_Str(Off);}
    // перевіряємо стан синього світлодіоду
       if ( (L & (1<<Blue_Led)) )   {USART_Send_Str(Blue); USART_Send_Str(On);}
    else   {USART_Send_Str(Blue); USART_Send_Str(Off);}
       USART_Send_Byte('\n');
}
// пошук та вибірка команди із  буферу
void Get_Commad(Ring_Buffer *pB, unsigned char *Com)
{
       unsigned char Byte = 0;
       if (pB->Counter>3) // перевіряємо чи в буфері достатньо символів для аналізу
       {
```

```
              while (pRxBuffer->Counter)// шукаємо перший символ
              {
                     Byte = GetByteBuffer(pB);  // вибираємо байт із вхідного буфера
                     if (Byte == 'c') // після першого символу йде 2 символи команди
                     {  // поміщаємо команду в масив для команди
                            Com[0]=GetByteBuffer(pB);
                            Com[1]=GetByteBuffer(pB);
                            New_Comm_F=1; // встановлюємо прапорець, що є необроблена команда
                            break; //команда знайдена то виходимо із циклу
                     }
              }
       }
}
// виконання команди
void Command_Execute(unsigned char *Com)
{
  if (New_Comm_F) // перевіряємо чи є нова команда в
  {
   New_Comm_F = 0;// команду виконали
// Дешифровка команди виконуємо за допомогою умовних операторів
   switch (Com[0]) {
     case 'R': {
       // другий символ може бути тільки '0', '1',
       // якщо ні то повідомлення про незрозумілу команду
       if (Com[1] > '1') { USART_Send_Str(WTF);}
        else {
             if (Com[1] == '1') {RGB_On(Red_Led);USART_Send_Str(Red);  USART_Send_Str(On); }
             if (Com[1] == '0') {RGB_Off(Red_Led);USART_Send_Str(Red); USART_Send_Str(Off);}
              }
                } break;

    case 'G':
     {
       if (Com[1] > '1')    { USART_Send_Str(WTF);}
        else {
         if (Com[1] == '1') {RGB_On(Green_Led); USART_Send_Str(Green); USART_Send_Str(On); }
         if (Com[1] == '0') {RGB_Off(Green_Led);USART_Send_Str(Green); USART_Send_Str(Off);}
              }
       } break;

    case 'B':
     {
       if (Com[1] > '1')    { USART_Send_Str(WTF);}
        else {
         if (Com[1] == '1') {RGB_On(Blue_Led);  USART_Send_Str(Blue); USART_Send_Str(On); }
         if (Com[1] == '0') {RGB_Off(Blue_Led); USART_Send_Str(Blue); USART_Send_Str(Off);}
              }
       } break;

       case 'S': {  Get_Status();} break;

       default: USART_Send_Str(WTF); // команда не зрозуміла
    } // кінець оператора switch
       // в масив команди записуємо '0'
              Command[0]=0x30;
              Command[1]=0x30;
       }
}
//-------------------- Основна програма -----------------------------

int main(void)
{
       // перед використанням буфери очищаємо
       ClearBuffer(pRxBuffer);
       ClearBuffer(pTxBuffer);

// ініціалізація портів для світлодіодів
```

```
        Port_Init(W_Led);
        RGB_Init();
        USART_Init(BAUDRATE); // початкове налаштування USART
        sei(); // дозволяємо переривання

        USART_Send_Str(Ready);
// головний цикл програми
    while (1)
        {
         Get_Commad(pRxBuffer, Command); // отримуємо команду із буфера
         Command_Execute(Command); // виконуємо команду
         Blink(); // фонова задача
    }
}
```

У головному циклі програми виконуються всього 3 задачі: вибірка команди, виконання команди та фонова задача - `Blink();`.

Завдання до лабораторної роботи.

1. Створити проект в Atmel Studio 7, надрукувати програму наведену в лістингах 4.4 – 4.6. Вивчити логіку роботи програми за допомогою симулятора.

   Розробити алгоритм, написати програму, відладити роботу за допомогою симулятора та перевірити за допомогою лабораторного стенду:

2. Модифікувати програму usart_ring_buffer та «Управління RGB світлодіодом за допомогою ШІМ»:

- число ШІМ задається через USART за допомогою команди для кожного кольору, це ж число та назва кольору виводиться на 7-сег. LED дисплей.
- при натисканні на кнопку 1, на дисплей виводиться значення виводиться в 1-й розряд назва кольору (R, G, B) інші 3 розряди виводиться число яке задає шпаруватість ШІМ для цього кольору. При повторному натисканні виводиться наступний колір.
- При натисканні на кнопку 2 на ПК надсилається звіт у вигляді:
  Red 123
  Green 123
  Blue 123
- передбачити команду від ПК, за якою надсилається звіт.

Контрольні запитання:

1. Стандарт послідовної передачі даних RS-232, логічні рівні, формат кадру, синхронний та асинхронний спосіб передачі.
2. Контрольні та статусні регістри модуля USART в ATmega328P.
3. Переривання, які генерує USART в ATmega328P.
4. Методика налаштування роботи вбудованого USART в ATmega328P.
5. Функції для прийому та передачі 1 байту.
6. Кільцевий буфер. Поясніть принцип використання кільцевих буферів, для обміну даними на прикладі програми usart_ring_buffer.

Література

1. Datasheet ATmega328P. http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf
2. Программирование микроконтроллеров ATMEL на языке С. Прокопенко В.С., К.: «МК-Пресс», 2012. – 320с.
3. Програмирование на языке С для AVR и PIC микроконтролеров. Шпак Ю.А., К.: «МК-Пресс», 2016. – 544стр.
4. Микроконтроллеры AVR семейств Tiny и Mega фирмы ATMEL. – М.: Издательский дом «Додэка-XXI», 2004. – 560 с.

5. Application Note_1497 AVR035: Efficient C Coding for 8-bit AVR microcontrollers. http://ww1.microchip.com/downloads/en/AppNotes/doc1497.pdf
6. Application Note_AVR42787 AVR Software User Guide http://ww1.microchip.com/downloads/en/AppNotes/Atmel-42787-AVR-Software-User-Guide_ApplicationNote_AVR42787.pdf
7. Програма Terminal 1.9. Інструкція до використання: https://faq.radiofid.ru/knowledge-bases/2/articles/13-kak-polzovatsya-terminalnoj-programmoj

Лабораторна робота 5.

Аналогово-цифрове та цифро-аналогове перетворення.

Мета: навчитися використовувати вбудований в ATmega328P аналогово-цифровий перетворювач та ознайомитися із цифро-аналоговим перетворювачем R-2R. Дослідити характеристики R-2R цифро-аналогового перетворювача.

Аналогово-цифровий перетворювач – це пристрій, який перетворює вхідний аналоговий сигнал (струм чи напруга) в цифровий сигнал (дискретний код), який кількісно характеризує величину вхідного сигналу. Це дає змогу МК працювати із різними давачами які перетворюють фізичну величину в електричний сигнал, наприклад: потужність світлового потоку перетворюється в напругу за допомогою фотодіоду, давач температури генерує напругу пропорційну до температури тощо.

Розглянемо принцип аналогово-цифрового перетворення. Нехай, маємо аналоговий сигнал - напругу, яка може неперервно змінюватися в межах від 0 до U_{max} протягом інтервалу часу τ. Для отримання цифрового сигналу (набору чисел) потрібно виконати декілька операцій: дискретизація, квантування та кодування аналогового сигналу.

Дискретизація – це представлення неперервного в часі аналогового сигналу послідовністю його значень (відліків). Ці значення вибираються через однакові інтервали часу (рис.1), які називають інтервал дискретизації T. Величину, $f_S = 1/T$, яка обернена до T називають частотою дискретизації. Чим вище частота дискретизації, тим точніше можна буде відтворити аналоговий сигнал, але висока частота обмежена можливостями аналогово-цифрового перетворювача (АЦП). Мінімальна частота, яка дозволяє відтворити аналоговий сигнал без спотворень визначається критерієм Котельникова-Найквіста:

$$f_S \geq 2 f_{max}, \tag{5.1}$$

де f_{max} - максимальна частота сигналу. Як правило частоту дискретизації вимірюють в кількості відліків АЦП за секунду, наприклад 15 kSPS (Samples per Second)

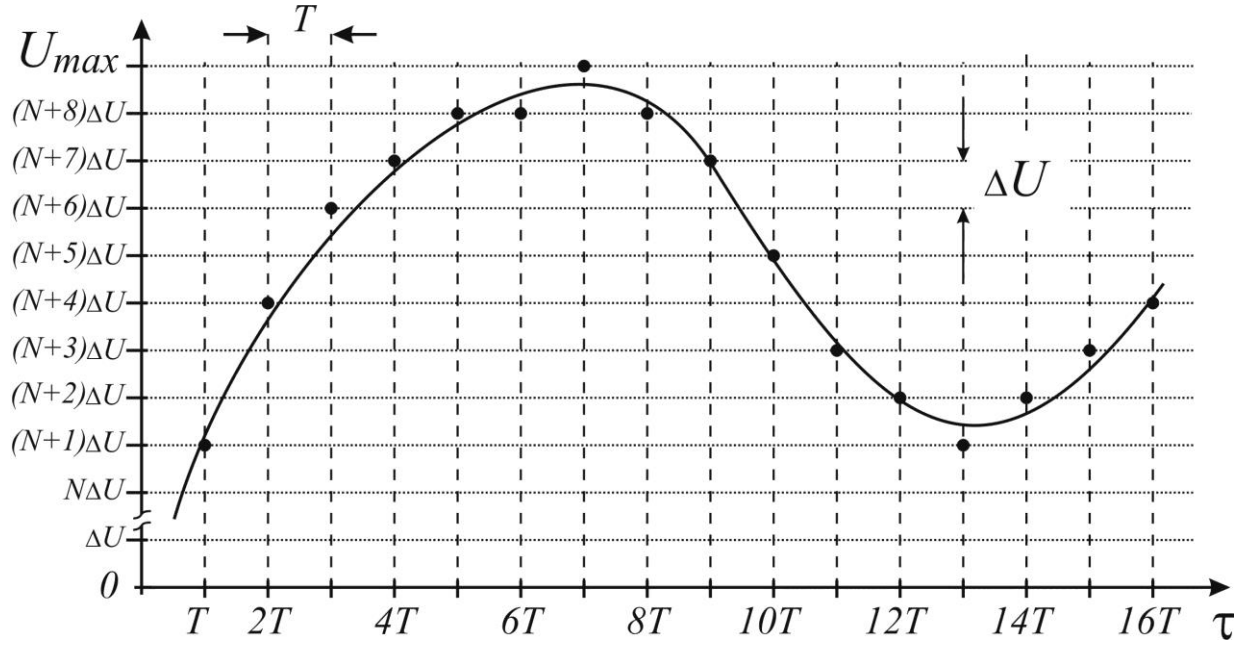


Рис. 1. Аналогово-цифрове перетворення, де T - інтервал дискретизації. ΔU - крок квантування, точками позначені відліки.

Квантування – це представлення (заміна) значення аналогового сигналу найближчим значенням із фіксованого набору чисел. Для того, щоб представити аналогову величину у вигляді числа, потрібно визначити мінімальну величину ΔU із якою ця величина буде

порівнюватися. Ця величина ΔU називається крок квантування, при заміні напруги на ΔU, вихідний код зміниться на 1 тому часто ΔU називають напруга найменш значущого біту – *LSB* (Last Significant Bit). Використовуючи крок квантування, поділимо весь діапазон на $U_{\max}$ інтервали значень кратні до кроку квантування – отримаємо скінченний набір рівнів квантування. Як правило, максимальна кількість рівнів квантування виражається степенями двійки, якщо $U_{\max} / \Delta U = 1024 = 2^{10}$ тобто, щоб представити це число потрібно 10 біт, тоді говорять, що роздільна здатність такого АЦП рівна 10-біт. Роздільна здатність буде тим більшою, чим більше біт потрібно для представлення результату:

$$\Delta U = 1\,LSB = \frac{U_{\max}}{2^N} \tag{5.2}$$

В реальних АЦП $U_{\max}$ задається значенням опорної напруги U_{ref}, яка може приймати різні значення, це призведе до зміни ΔU, тому при описі характеристик АЦП всі величини наводять не в вольтах , а в напрузі LSB.

Величину аналогового сигналу можна представляти в межах кожного інтервалу числом, яке рівне до порядкового номеру рівня квантування – округляємо до значення кратного до ΔU. Очевидно, що при округленні значень виникає похибка квантування, яка буде рівна $\pm\frac{1}{2}\Delta U$.

Похибка квантування є невід'ємною частиною аналогово-цифрового перетворення. Її неможливо уникнути і, оскільки вона залежить тільки від розрядності АЦП, то вона буде однаковою для різних АЦП однакової розрядності.

Таким чином, із аналогового сигналу (який може приймати будь-які значення в межах інтервалу) отримали скінченний набір чисел. Тепер отриману сукупність чисел потрібно виразити за допомогою комбінації деяких знаків (символів) чи цифр. Сукупність знаків (символів) і набір правил за яким число представляється у вигляді впорядкованого набору знаків називають кодом. Скінченна сукупність знаків, яка відповідає певному числу називають кодовим словом. Операція кодування – це представлення цифрового (квантованого) сигналу у вигляді послідовності кодових слів. Кодування можна проводити за будь-якою системою у двійковому коді, в десятковій системі числення, у двійково-десятковому коді, тощо.

Для зворотного перетворення сукупності чисел (цифровий сигнал) в пропорційну до них напругу чи струм (в аналоговий сигнал) використовують цифро-аналогові перетворювачі. Аналоговий сигнал, який отримується в результаті цифро-аналогового перетворення є ступінчастий сигнал із певним значенням можливих амплітудних значень. Величина сходинки буде залежати від величини кроку квантування та розрядності цифрового сигналу, тобто чим більше біт в ньому тим менша величина сходинок. Для отримання неперервного аналогового сигналу його потрібно пропустити через фільтр нижніх частот.

Похибки аналогово-цифрового та цифро-аналогового перетворення. Перетворення аналогового сигналу в цифровий і навпаки виконується за допомогою пристроїв, які, як правило, являють собою інтегральні мікросхеми і називаються аналогово-цифровий перетворювач та цифро-аналоговий перетворювач (ЦАП). Для правильного використання цих пристроїв потрібно знати їх характеристики та параметри. Розглянемо на прикладі АЦП. Основною характеристикою АЦП є передавальна характеристика – це залежність між числовим еквівалентом (кодом) цифрового сигналу та відповідного до нього аналогового сигналу. Передавальна характеристика ідеального АЦП та ЦАП є ступінчаста функція із 2^N сходинок (рис. 2, а), де N – розрядність АЦП. Кожний горизонтальний відрізок цієї функції відповідає одному із значень вихідного коду АЦП, якщо з'єднати лініями початки цих горизонтальних відрізків то ідеальна передавальна характеристика буде прямою лінією яка проходить через початок координат.

Точність буде визначатися відхиленням передавальної характеристики реального АЦП від ідеальної. Точність АЦП не є еквівалентною до роздільної здатності, яка буде визначатися розрядністю АЦП (хоча часто ці параметри путають) і рівна до кроку квантування 1 *LSB*

(4.2). Роздільна здатність характеризує теоретично можливу мінімальну величину похибки, а реальні похибки характеризуються сумарним відхиленням передавальної характеристики АЦП від ідеальної. Параметри, які характеризують це відхилення від ідеальної поведінки: адитивна похибка (Offset error), мультиплікативна похибка (Gain error), диференціальна нелінійність (DNL Differential Non-linearity), інтегральна нелінійність (INL Integral Non-linearity).

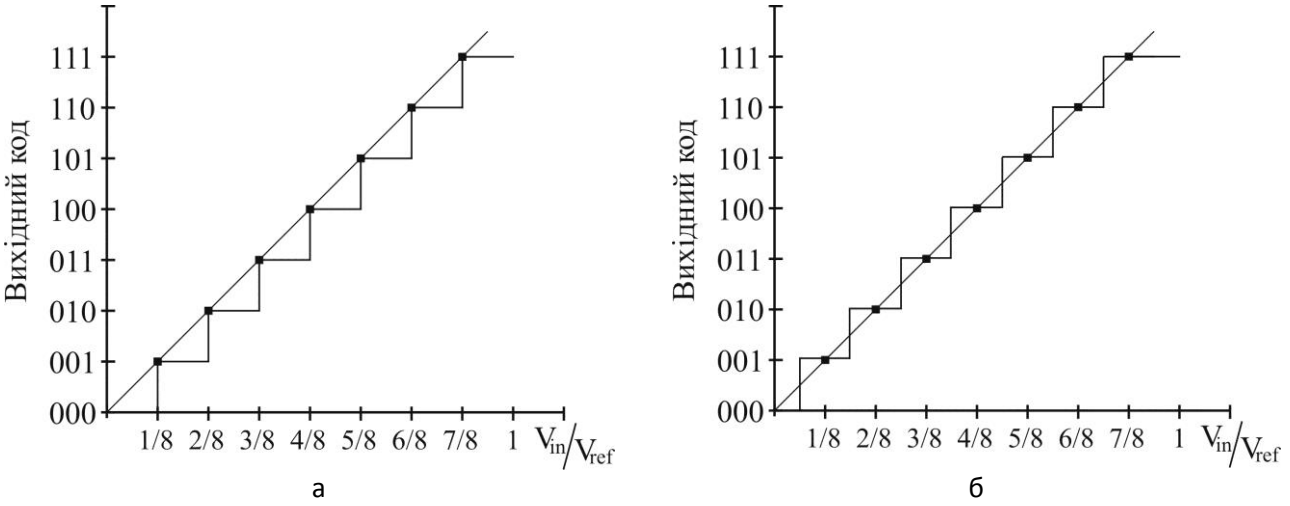


Рис. 2. Передавальна характеристика: а) ідеального 3-розрядного АЦП; б) ідеального 3-розрядного АЦП із зміщенням передавальної характеристики на $-\frac{1}{2}LSB$.

Адитивна похибка. Ідеальна передавальна характеристика АЦП перетинає початок координат, а перша зміна коду від 0 до 1 відбувається при досягненні вхідної напруги значення $1\ LSB$. Адитивна похибка є зміщення всієї передавальної характеристики ліворуч чи праворуч від початку координат і буде визначатися відхиленням напруги першого переходу коду 0→1 від значення напруги $1\ LSB$ (рис.3, а).

Мультиплікативна похибка – це різниця між ідеальною та реальною передавальною характеристикою в точці максимального вихідного значення, за умови компенсованої адитивної похибки (відсутнє початкове зміщення). Це проявляється, як зміна нахилу передавальної характеристики (рис. 3, б).

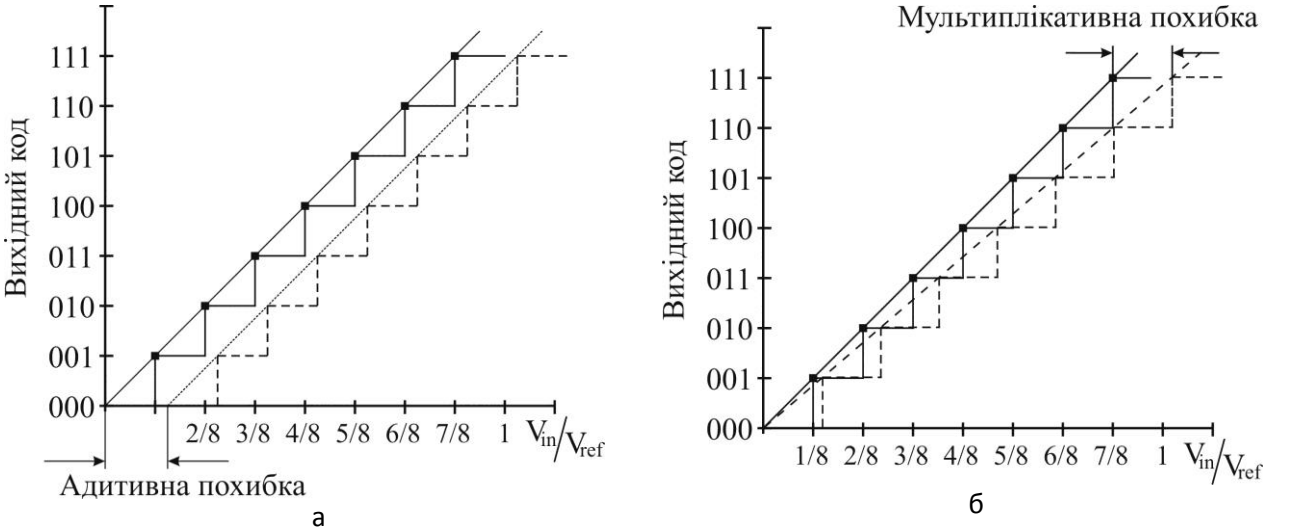


Рис. 3. Адитивна (а) та мультиплікативна (б) похибки АЦП. Суцільна лінія - передавальна характеристика ідеального АЦП, пунктирна – реальна.

Диференціальна нелінійність. У ідеальної передавальної характеристики, ширина кожної сходинки повинна бути однакова і рівна значенню напруги $1\ LSB$. Це відповідає тому, що перехід до наступного значення коду буде відбуватися при збільшенні вхідної напруги на $1\ LSB$. Тобто, якщо вихідний код відрізняється на 1, то і різниця вхідних напруг

буде відрізнятися на 1 *LSB*. Відхилення цієї різниці від значення 1 *LSB* буде визначати диференціальну нелінійність АЦП (рис.4, а).

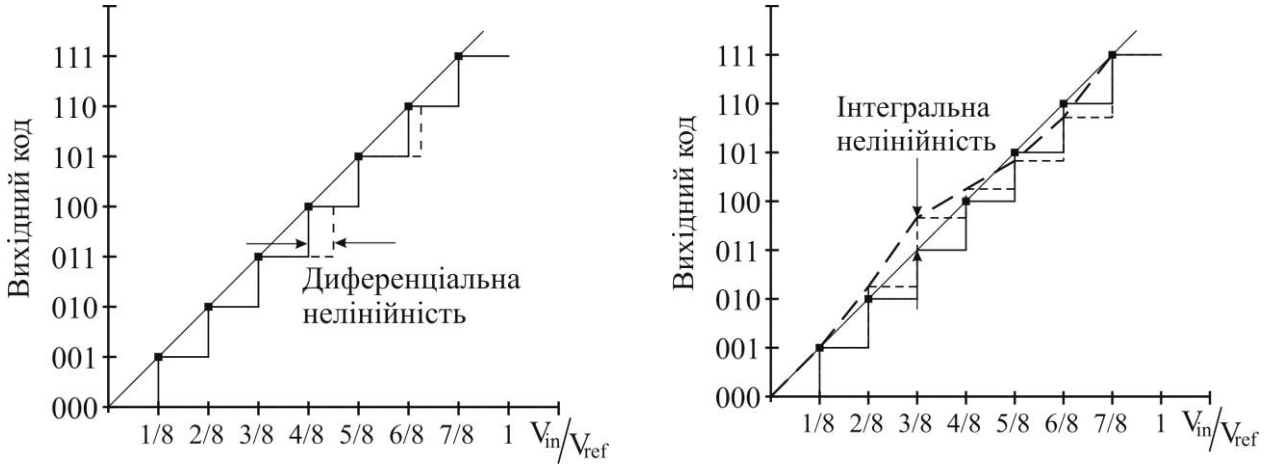


Рис. 4. Похибки АЦП: а) диференціальна нелінійність; б) інтегральна нелінійність. Суцільна лінія - передавальна характеристика ідеального АЦП, пунктирна – реальна.

Інтегральна нелінійність – це похибка, яка визначається, як максимальне відхилення передавальної характеристики реального АЦП (після усунення адитивної та мультиплікативної похибок) від лінійної передавальної характеристики ідеального АЦП (рис.4, б).

Вбудований АЦП в ATmega328P має такі характеристики:

- 10-біт роздільна здатність;
- 0,5 *LBS* інтегральну нелінійність;
- ±2 абсолютна точність;
- час перетворення 13 – 260 мкс;
- максимальна швидкість перетворення 76,9 kSPS (15 kSPS за максимальної роздільної здатності;
- 8 канальний мультиплексор вхідних сигналів
- можливість підключатися до вбудованого давача темпратури;
- діапазон вхідної напруги $0-V_{CC}$;
- можливість підключення вбудованого джерела опорної напруги 1,1 В;
- наявність переривання по закінченню аналогово-цифрового перетворення;
- режими роботи: однократне перетворення, постійне автоматичне перетворення;
- наявність окремої лінії живлення AV_{CC} для зменшення завад.

Детальний опис АЦП наведений в технічному описі мікроконтролера ATmega328P, в розділі 28, ст.305.

Налаштування робочих режимів АЦП відбувається за допомогою контрольних регістрів (детальний опис розділ 29.9 ст. 316 «Опис регістрів»):

ADMUX – регістр вибору джерела опорної напруги та вхідного каналу;

ADCSRA – контрольний регістр А;

ADCL, ADCH – два регістри даних, в них поміщається результат аналогово-цифрового перетворення;

ADCSRB – контрольний регістр В;

DIDR0 (Digital Input Disable Register 0) – регістр відповідальний за відключення вхідних цифрових буферів.

Детальний опис регістрів буде наведено нижче.

Початок перетворення. Аналогово-цифрове перетворення починається, якщо записати «1» в біт **ADSC** (ADC start conversion) регістру **ADCSRA**, після закінчення перетворення цей біт автоматично скидається в «0». Це дозволяє відслідковувати момент закінчення перетворення. Після закінчення перетворення результат буде знаходитися в регістрах **ADCL, ADCH.** Аналогово-цифрове перетворення може бути запущене

автоматично різними подіями, якщо встановити біт **ADATE** в **ADCSRA.** Вид події буде визначатися налаштуванням регістру **ADCSRB,** це може бути зовнішнє переривання INT0, переривання від аналогового компаратора та переривання від таймерів-лічильників TC0, TC1.

Швидкість перетворення та тактування АЦП. Для коректної роботи, АЦП потрібна тактова частота в діапазоні 50 – 200 кГц. Таку тактову частоту можна отримати із тактової частоти МК, використовуючи вбудований подільник частоти. Коефіцієнт ділення частоти буде визначатися комбінацією управляючих бітів в регістрі **ADCSRA.** Подільник частоти включається на початку перетворення і вимикається після його закінчення. Перше перетворення, після включення АЦП триває 25 тактів, всі подальші перетворення відбуваються протягом 13 тактів. Після закінчення перетворення, результат поміщається в регістри **ADCL, ADCH** і встановлюється прапорець переривання «закінчення перетворення» (ADC - AD conversion complete) **ADIF** в регістрі **ADCSRA.**

Вибір вхідного каналу та опорної напруги АЦП. Вибір вхідного каналу та джерела опорної напруги відбувається шляхом встановлення бітів в регістрі **ADCSRB.** В режимі однократного перетворення, зміну вхідного каналу потрібно виконувати перед його початком, оскільки під час перетворення регістр **ADCSRB** заблокований для запису. Якщо АЦП працює в автоматичному режимі, то для гарантованої і однозначної зміни каналу потрібно зупинити АЦП: біти ADEN, ADATE = 0 в регістрі **ADCSRA.**

Результат аналогово-цифрового претворення. Результат знаходиться в парі регістрів **ADCL, ADCH,** до яких можна звертатися, як до одного 16-ти розрядного регістру **ADC.** Число, яке є в цьому регістрі, пов'язане із вхідною та опорною напругою:

$$ADC = V_{in}\frac{1024}{V_{ref}} \quad , \tag{5.3}$$

де V_{in} - вхідна напруга, V_{ref} - опорна напруга.

Значення в регістрі **ADC** рівне 0 відповідає нульовому потенціалу (аналогова земля), а 1023 відповідає опорній напрузі.

Вимірювання температури. В ATmega328P є вбудований давач температури, який за допомогою мультиплексору може бути підключений до АЦП. Для підключення температурного давача потрібно встановити в регістрі ADMUX біти MUX[3..0] в 1000 та підключити внутрішнє опорне джерело напруги в 1,1 В. Виміряна напруга лінійно пов'язана з температурою із чутливістю $1 mB/^{\circ}C$, точність вимірювання температури $\pm 10\,^{\circ}C$. Типове значення вихідної напруги температурного давача для деяких температур:

температура	$-45\,^{\circ}C$	$+25\,^{\circ}C$	$+85\,^{\circ}C$
напруга	242 мВ	314 мВ	380 мВ

Опис регістрів.

ADMUX (ADC Multiplexer Selection Register)

bit	7	6	5	4	3	2	1	0
ADMUX	REFS1	REFS0	ALAR	-	MUX3	MUX2	MUX1	MUX0

REFS1, REFS0 (Reference Selection) – ці біти відповідають за вибір опорної напруги:

REFS[1..0]	Опорна напруга
00	AVREF, внутрішня V_{ref} відключена
01	AV_{CC}, із зовнішнім конденсатором біля AVREF виводу
10	-
11	Внутрішня $V_{ref} = 1{,}1\,B$ із зовнішнім конденсатором біля AVREF виводу

ALAR – визначає розташування даних в регістрі ADC. Якщо ALAR=1, то дані розташовуються від старшого до 8-го біту в **ADCH,** а два наймолодші біти розташовані в

старших розрядах **ADCL.** Якщо ALAR=0 то біти з 0 по 7 розташовані в **ADCL,** а два старші біти в молодших розрядах регістру **ADCH.**

MUX[3..0] – визначають номер каналу вхідного мультиплексора АЦП.

MUX[3..0]	Номер каналу
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000	Температурний давач
1110	Внутрішнє джерело опорної напруги $V_{ref} = 1{,}1\,B$
1111	0 В (сигнал GND – земля)

ADCSRA (ADC Control and Status Register A)

bit	7	6	5	4	3	2	1	0
ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

ADEN (ADC enable) – встановлення цього біту вмикає АЦП, його очищення вимикає АЦП.

ADSC (ADC Start Conversion). В режимі одноразового перетворення встановлення ADSC=1 запускає процес аналогово-цифрового перетворення. В режимі постійного перетворення ADSC=1 запускає перше перетворення. Після закінчення перетворення автоматично скидається в 0.

ADATE (ADC Auto Trigger Enable). Якщо ADATE=1,то дозволений автозапуск аналогово-цифрового перетворення за визначеною подією. Вид події визначається налаштуванням в регістрі ADCSRB.

ADIF (ADC Interrupt Flag) – прапорець переривання «закінчення перетворення». Після закінчення перетворення, ADIF автоматично встановлюється і формується запит на переривання, якщо переривання дозволені, то після виконання обробника переривань ADIF скидається в 0 автоматично. Можна скинути програмно, якщо записати в цей біт 1.

ADIE (ADC Interrupt Enable). Якщо цей біт встановлений, то переривання по закінченню перетворення буде дозволене.

ADPS[2..0] – визначають коефіцієнт ділення системної частоти *F_CPU* для формування тактового сигналу для АЦП.

ADPS[2..0]	Коефіцієнт ділення частоти
000	2
001	2
010	4
011	8
100	16
101	32
110	64
111	128

ADCSRB (ADC Control and Status Register B)

bit	7	6	5	4	3	2	1	0
ADCSRB	-	ACME	-	-	-	ADTS2	ADTS1	ADTS0

ACME – використовується для управління аналоговим компаратором.
ADTS[2..0] – встановлює джерело для автоматичного запуску АЦП.

ADTS [2..0]	Коефіцієнт ділення частоти
000	Вільний режим роботи
001	Аналоговий компаратор
010	Зовнішнє переривання INT0
011	Таймер-лічильник TC0 переривання по співпадінню A
100	Переривання переповнення TC0
101	Таймер-лічильник TC1 переривання по співпадінню B
110	Переривання переповнення TC1
111	TC1 – подія «захват значення TCNT1»

DIDR0 (Digital Disable Register 0).

bit	7	6	5	4	3	2	1	0
DIDR0	ADC7D	ADC6D	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D

Якщо відповідний біт встановлений в 1 то вхідний цифровий буфер відповідного входу ADC буде відключений. Відповідний PINn регістр буде читатися як 0, незалежно від значення сигналу на вході. Відключення вхідних буферів зменшує енергоспоживання МК.

Застосування АЦП.

Перед застосуванням АЦП потрібно його правильно налаштувати, для цього потрібно визначити такі величини:

- розмах вхідної напруги: від цього залежить вибір джерела опорної напруги і, відповідно, величина роздільної здатності (величина кроку квантування). Як правило найчастіше використовується 2 значення опорної напруги: для отримання максимального діапазону вхідних напруг $V_{ref} = AV_{CC}$. Для отримання кращої стабільності використовують внутрішнє джерело опорної напруги $V_{ref} = 1{,}1\,B$.
- номер вхідного каналу, якщо кількість вхідних сигналів не більше 2 то краще використовувати канали ADC6, ADC7, які не суміщені із лініями паралельних портів введення-виведення.
- роздільна здатність 10-біт чи 8-біт, іноді достатньо лише 8-ми бітної роздільної здатності;
- режим роботи: однократне перетворення, вільний робочий режим, автоматичний запуск перетворення за визначеною подією.
- робоча частота АЦП – задається подільником частоти так, щоб робоча частота не виходила за межі оптимальної в 50 – 200 кГц.

Приклад функції ініціалізації вбудованого АЦП в ATmega328P та функції, яка виконує одноразове аналогове-цифрове перетворення наведено в лістингу 5.1.

Лістинг 5.1. Приклад ініціалізації та використання АЦП в режимі одноразового перетворення.

```
void ADC_Init(unsigned char ADC_Channel) // ініціалізація АЦП
{
// джерело опорної напруги Vref внутрішне 1,1 В
  ADMUX=(1<<REFS0)|(1<<REFS1);
  ADMUX|=ADC_Chanal; // встановлюємо номер каналу
// включаємо АЦП, подільник частоти 128 f=F_CPU/128=125 kHz, (F_CPU=16 МГц)
  ADCSRA=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
  ADCSRA|=(1<<ADSC);  // запускаємо 1-ше перетворення
  while (ADCSRA & (1<<ADSC));// чекаємо на його закінчення
}
//*********************************************************************
int Get_ADC(void)    // функція, яка виконує аналог.-цифр. перетворення
{
       ADCSRA|=(1<<ADSC);            // запускаємо перетворення
       while (ADCSRA & (1<<ADSC));  // чекаємо на його закінчення
       return ADC;                              // результат в регістрі ADC
}
```

Налаштування роботи АЦП відбувається в функції `void ADC_Init(unsigned char ADC_Channel)`, після налаштування (встановлення $V_{ref} = 1{,}1\,B$, номеру вхідного каналу, робочої частоти та включення АЦП) запускаємо одноразове перетворення для того, щоб всі частини модуля АЦП увімкнулися – це займає 25 циклів тактової частоти АЦП. Функція, яка виконує аналогово-цифрове перетворення `int Get_ADC(void)` доволі проста – запускаємо одноразове перетворення та чекаємо на його закінчення. Недоліком цієї функції є те, що витрачається час процесора на очікування закінчення перетворення. Позбавитися цього можна, використавши автозапуск перетворення.

Розглянемо приклад використання автозапуску аналогово-цифрового перетворення. Нехай МК повинен виконувати 2 задачі: 1) періодично вимірювати температуру, відображати її на семисегментному дисплеї; 2) моргати світлодіодом із частотою 2 Гц. Побудуємо термометр використавши вбудований АЦП та аналоговий давач температури L35. АЦП налаштуємо на автозапуск, а результат аналогово-цифрового перетворення будемо обробляти в перериванні “ADC complete” – закінчення аналогово-цифрового перетворення. Давач температури L35, відкалібрований в градусах Цельсія – за температури $0\,^{\circ}C$ вихідна напруга рівна $0\,мВ$ він має лінійну шкалу із чутливістю $+10\,мВ/^{\circ}C$ та похибкою в $\pm 0{,}5\,мВ/^{\circ}C$. Тобто, за температури $20\,^{\circ}C$ вихідна напруга буде рівна $200\,мВ$. Вимірювання будемо проводити із використанням внутрішнього джерела опорної напруги 1,1 В. Давач температури під’єднаємо до входу AD7, а вимірювання будемо проводити із інтервалом в 1 мс. Результат будемо усереднювати по 128 вимірюваннях ($128 = 2^7$, при обчисленні середнього значення ділення на 128 буде замінено на операцію зсуву на 7 позицій). Аналогово-цифрове перетворення буде запускатися автоматично за подією «переривання по співпадінню В таймера-лічильника ТС1». Програма наведена в лістингу 5.2, логіка програми пояснена в коментарях.

Лістинг 5.2. Вимірювання температури за допомогою вбудованого АЦП.

```
/*
 * lab5-1.c
 *Термометр
 */
/* ************************************************************
/        7-сегментний світллодіодний дисплей приєднаний до S7LED_Port:
/   S7LED_Port's bit     |7 |6|5|4|3|2|1|0|
/   7 seg LED            |dp|g|f|e|d|c|b|a|
/
/  7-сегментний світллодіодний дисплей із загальним анодом -
/  сегменти будуть світитися, якщо рівень напруги - 0.
/
/ ************************************************************ */
#define F_CPU 16000000UL   // Частота роботи ATmega232P 16 МГц

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

#define LED_Button_PORT PORTC   // порт, до якого буде приєднано  кнопка
#define LED_Button_DDR  DDRC      //  регістр вибору напрямку роботи порта кнопки
#define Button_PIN PINC         // регістр для перевірки стану кнопки
#define LED_1 PC4          // світлодіод для фонової задачі

#define S7LED_Port PORTD    // 7-сегментний дисплей приєднаний до PORTD
#define S7LED_DDR DDRD      // регістр задає режим роботи (вхід-вихід) LED_7seg_Port

#define Anode_Port      PORTB //  аноди 7-ми сегментного індикатора приєднати до PORTB
#define Anode_Port_DD   DDRB
#define dp 7        //десяткова кома

void Int_Port(void);      // початкова ініціалізація портів
unsigned char Chk_Kn(char k);   // перевірка стану кнопки
```

```
void Init_Timer1(void);           // ініціалізація таймера-лічильника TC0
void ADC_Init(unsigned char AD_Channel);

// функція, яка перетворює 10-ве число Data в набір цифр: 1234 - "1", "2", "3", "4", які
// поміщаються в масив Number_String
void Int_To_Array( unsigned char* Number_String, int Data);

// Глобальні змінні
// кодування цифр в 7-сег. код:
unsigned char Digits[11]={0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8, 0x80, 0x90, 0xFF};
unsigned char Number_String[4] = {0}; //масив для збереження числа у вигляді сукупності цифр

//лічильник, в якому зберігається поточне значення номеру цифри числа яке виводиться на 7LED
unsigned char i = 0;
unsigned int n = 0;

// змінна для накопичення результату аналогово-цифрового перетворення
unsigned long int Summa = 0;
unsigned char k = 0;       // поточне значення кількості вимірювань

// змінна, де буде поміщено усереднене значення аналогово-цифрового перетворення
unsigned int Result = 0;
unsigned int mVolt = 0;    // результат АЦ перетворення переведений в мВ

//---------- Опис функцій --------------------------
void Int_Port(void)
{
 LED_Button_DDR |= (1<<LED_1); // лапка для світлодіода - вихід

 S7LED_DDR = 0xFF;     // порт, до якого підключений 7-ми сег. дисплей - вихід
 S7LED_Port = 0xFF;     // всі лінії порту  1 - дисплей вимкнений

 Anode_Port_DD = 0xFF;  //  порт до якого приєднані аноди 7LED - вихід
 Anode_Port = 0x00;    // на аноди подаємо 0 В
}
/*
   перетвоюємо ціле число Data в упорядкований набір цифр, який зберігається в масиві
Number_String у вигляді: кількість тисяч, кількість сотень, кількість десятків, кількість
одиниць, ці цифри будуть використані, як індекси для виборки 7LED коду цифри із масиву
Digits
*/
void Int_To_Array( unsigned char* Number_String, int Data)
 {// обнуляємо попереднє значення
  Number_String[0] = Number_String[1] = Number_String[2] = Number_String[3] = 0;
        i=0; // відображення цифри починаємо із початку
     while (Data>=1000)              // вичисляємо кількість тисяч
     { Data-=1000; Number_String[3]++; }
     while (Data>=100)              // вичисляємо кількість сотень
     { Data-=100; Number_String[2]++; }
    while (Data>=10)                    // вичисляємо кількість десятків
     { Data-=10; Number_String[1]++; }
     Number_String[0]+=Data;        // те що лишилося - кількість одиниць

 // якщо старші розряди 0, то їх не відображаємо
    if(Number_String[3] == 0) Number_String[3] = 10;
     {
       if (Number_String[3] == 10 && Number_String[2] == 0) {Number_String[2] = 10;}
       if (Number_String[3] == 10 && Number_String[2] == 10 && Number_String[1] == 0)
       { Number_String[1] = 10;}
     }
}
/* Ініціалізація таймера TC1 - генерація запиту на переривання при співпадінні вмісту
регістрів TCNT1 та OCR1A, OCR1B
   переривання OCM1B буде запускати аналогово-цифрове перетворення кожні 1 мс
   переривання OCM1A буде обслуговувати динамічну індикацію кожні 3,2 мс */
void Init_Timer1()
{
```

```
       TCCR1B |= (1<<WGM12); // вибираємо режим CTC при OCM1A
// Дозволяємо переривання співпадінні вмісту регістрів TCNT0 та OCR0A
       TIMSK1 |= (1<<OCIE1A);
       OCR1A=200; // інтервал часу буде визначатися OCR0A*256/16 000 000 = 3,2 мс
       OCR1B=63; // OCR0A*256/16 000 000 = 1,008 мс
       TIFR1 |=(1<<OCF1A)|(1<<OCF1B); // скидаємо прапорці перериваннь
       TCNT1=0x00; //записуємо початкове значення в лічильний регістр
       //включаємо лічильник та встановлюємо подільник частоти F_CPU/256
       TCCR1B |=(1<<CS12);
}

ISR(TIMER1_COMPA_vect) //динамічна індикація
{
  Anode_Port=0x00;       // аноди вимкнули, щоб погасити 7LED
// в порт, до якого приєднані катоди 7LED, вивели код поточної цифри
  if (i==1)
  {// ділення на 10 виконуємо переносом десяткової коми: в другій цифрі включаємо сегмент dp
        S7LED_Port=(Digits[Number_String[i]]) & (~(1<<dp));
  }
  else
    { S7LED_Port=(Digits[Number_String[i]]); }
       Anode_Port=(1<<i);  // включили відповідний анод
//лічильник цифри збільшили на 1, якщо вийшли за межі кількості розрядів то i=0
  if (++i>3) i=0;
}

void ADC_Init(unsigned char AD_Channel)
{

       ADMUX |=(1<<REFS1)|(1<<REFS0); // підключаємо внутрішнє джерело опорної напруги
       ADMUX |=AD_Channel; // встановлюємо номер каналу
// вмикаємо АЦП, вмикаємо автозапуск АЦП, подільник частоти 128 (f=125 kHz)
       ADCSRA=(1<<ADEN)|(1<<ADATE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
       // вибираємо джерело автозапуску АЦП подію "співпадіння значень в TCNT1 та OCR1B"
       ADCSRB |=(1<<ADTS2)|(1<<ADTS0);
       ADCSRA|=(1<<ADSC);             // запускаємо 1-ше перетворення
       while (ADCSRA & (1<<ADSC));   // чекаємо на його закінчення
       // вмикаємо переривання по закінченню  аналогово-цифрового перетворення
       ADCSRA |=(1<<ADIE)|(1<<ADIF);
}

ISR(ADC_vect) // переривання по завершенню аналогово-цифрового перетворення
{
       Summa+=ADC; //накопичуємо результат перетворення
       k++; // лічильник кількості перетворень збільшуємо на 1
  if (k==128) // якщо провели 128 перетворень
  {
// знаходимо середнє значення замість ділення на 128 використовуємо зсув на 7 позицій
   Result=Summa>>7;
// оскільки АЦП має 10 розрядів накладаємо маску, щоб виділити тільки перші 10 розрядів
   Result &=(0x3FF);
/*  Оскільки використовується внутрішнє VREF=1,1 V = 1100 mV, то значення напруги в mV:
                           Vin=Result*(1100/1024)
напряму таку формулу обчислити неможливо бо Result*1100>2^16, то виділимо цілу частину:
Vin=Result*(1100/1024)=Result*(1+19/256)=Result+(Result*19/2^8)=Result+(Result*19>>8) */
         mVolt=Result+((Result*19)>>8);
         Int_To_Array(Number_String, mVolt); // виводимо результат на дисплей
         k=0;
         Summa=0; // починаємо накопичувати знову
  }

  ADCSRA |=(1<<ADIF); // скидаємо прапорець переривання

/* оскільки, відсутня функція обробник переривання  "співпадіння значень в TCNT1 та OCR1B",
     прапорець переривання цієї події потрібно скидати програмно, інакше автозапуск АЦП
     відбудеться лише один раз */
  TIFR1 |=(1<<OCF1B); // скидаємо прапорець переривання
```

```
}

int main(void)
{
 Int_Port();
 ADC_Init(7); // Початкове налаштування АЦП
 Init_Timer1(); // Початкове налаштування таймера 1
  n=1234;
  sei(); // дозволяємо переривання
  Int_To_Array(Number_String, n);
  _delay_ms(1000);

/* в основному циклі програми виконується фонова задача, процеси перетворення, накопичення
та усереднення, відображення результату відбуватимуться за перериваннями. */
while (1)
{
 // фонова задача моргати світлодіодом
       Button_PIN |=1<<LED_1;
       _delay_ms(250);
}
}
```

Деякі пояснення до програми: аналогово-цифрове перетворення відбувається із інтервалом в 1 мс. Обробка результату відбувається в перериванні по закінченні аналогово-цифрового перетворення: результат усереднюється по 128 вимірюванням, значення перераховується із числа в напругу, яка діє на вході в мВ:

$$V_{in} = ADC\frac{V_{ref}}{1024} = ADC\frac{1100}{1024} = ADC\left(1+\frac{76}{1024}\right) = ADC + \frac{19\cdot ADC}{256} = ADC + (19\cdot ADC) >> 8 .$$

Оскільки у ATmega328P відсутній апаратний подільник, то формулу перетворили таким чином, щоб звести її до множення та зсуву. Оскільки чутливість температурного давача $+10\ мВ/^{\circ}C$, то для відображення температури отримане значення напруги потрібно ділити на 10, але ділення на 10 еквівалентно включенню десяткової коми після 2-го знаку.

В функції обробнику переривання `ISR(ADC_vect)`, обов'язково, потрібно скинути програмно прапорець переривання події, яка запустила перетворення: `TIFR1|=(1<<OCF1B);`, оскільки, відсутня функція обробник переривання "співпадіння значень в TCNT1 та OCR1B" прапорець переривання цієї події потрібно скидати програмно, інакше автозапуск АЦП відбудеться лише один раз.

Цифро-аналогове перетворення.

Для зворотного перетворення числа в напругу слугує цифро-аналогове перетворення . В лабораторній роботі №3 «Вивчення роботи таймерів-лічильників» описаний метод отримання аналогового сигналу із сукупності чисел – широтно-імпульсна модуляція. ШІМ можна вважати реалізацією цифро-аналогового перетворення. Окрім ШІМ існують інші методи побудови цифро-аналогових перетворювачів: цифро-аналогове перетворення із ваговими резисторами та резисторна матриця R-2R.

Розглянемо схему 4-бітного ЦАП із ваговими резисторами (рис. 5, а), на чотири входи D_3, D_2, D_1, D_0, які керують ключами, подається 4-х бітний цифровий сигнал. Номінали резисторів R_0, R_1, R_2, R_3 підібрані за ваговими коефіцієнтами відповідних двійкових розрядів:

$$R_n = \frac{R}{2^n},\ n = 0, 1, 2, 3 . \qquad (5.5)$$

Сумарний струм, який протікає через резистори буде залежати від поданого двійкового числа D_3, D_2, D_1, D_0 на вхід схеми і він створить падіння напруги на резисторі R_{out}. Вихідна напруга U_{out} буде пропорційна до значення двійкового числа, яке подане на вхід ЦАП:

$$U_{out} = R_{out}\left(D_3 \frac{U_{ref}}{R_3} + D_2 \frac{U_{ref}}{R_2} D_1 \frac{U_{ref}}{R_1} + D_0 \frac{U_{ref}}{R_0} \right) = U_{ref} \frac{R_{out}}{R} \left(D_3 2^3 + D_2 2^2 + D_1 2^1 + D_0 2^0 \right) \quad (5.6)$$

Така схема має суттєвий недолік: точність перетворення залежить від абсолютної точності виготовлення резисторів і це накладає обмеження на кількість розрядів та призводить до температурної нестабільності таких пристроїв. Цього недоліку позбавлений ЦАП, який побудований на основі матриці однакових резисторів R-2R, схема якого наведена на рис. 5, б.

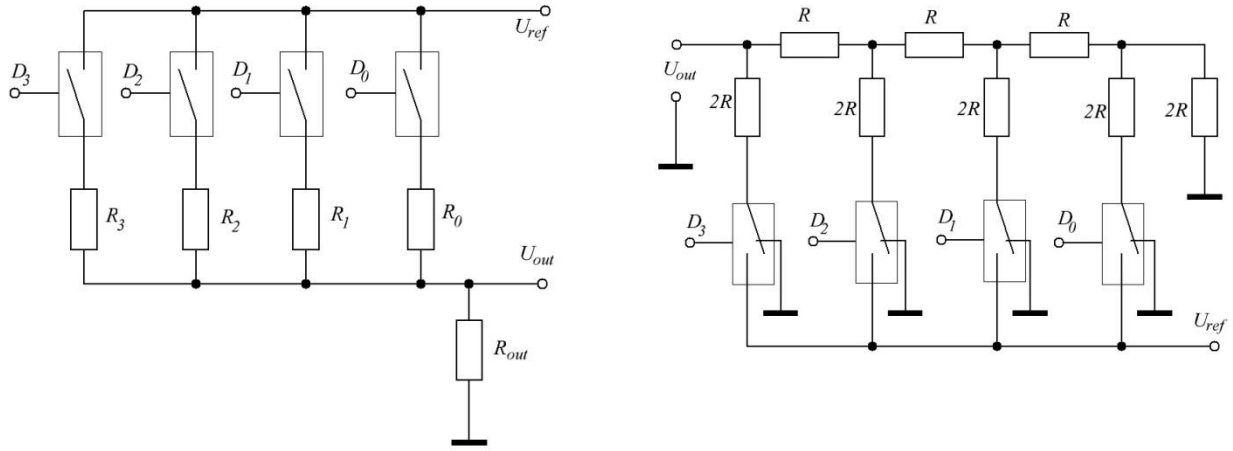


Рис. 5. Резистивні ЦАП: а) на основі вагових резисторів; б) резисторна матриця R-2R

Матриця R-2R працює, як сукупність подільників напруги, яка має постійний опір і кожна наступна ланка ділить напругу в два рази. Для 4-х розрядного ЦАП на вхід якого подане двійкове число D_3, D_2, D_1, D_0 вихідна напруга буде визначатися:

$$U_{out} = U_{ref}\left(D_3 \frac{1}{2^1} + D_2 \frac{1}{2^2} + D_1 \frac{1}{2^3} + D_0 \frac{1}{2^4} \right), \quad (5.7)$$

в загальному випадку, для *n* розрядного числа $D_{n-1}, D_{n-2}, \ldots D_1, D_0$:

$$U_{out} = U_{ref}\left(\frac{D_{n-1}}{2^1} + \frac{D_{n-2}}{2^2} + \ldots + \frac{D_1}{2^{n-1}} + \frac{D_0}{2^n} \right). \quad (5.8)$$

Похибки ЦАП R-2R. Для побудови ЦАП R-2R великої розрядності потрібна велика точність виготовлення резисторів. Наприклад: для 4-х бітного ЦАП точність повинна бути кращою за $1/2^4 = 1/32 \approx 3\%$, очевидно, що із використанням стандартних 1% дискретних резисторів не можна побудувати ЦАП більше 5 розрядів, а для 8-бітного ЦАП потрібна точність $1/2^8 = 1/256 \approx 0,4\%$.

Розглянемо приклад використання R-2R ЦАП для генерації трикутного сигналу. Для цього із певною періодичністю будемо записувати в порт число від 0 до 255, а потім від 255 до 0. Період між кожним записом будемо відраховувати за допомогою таймера-лічильника TC0. Використаємо переривання «співпадіння TCNT0 та OCR0A». Програма наведена в лістингу 5.3.

Лістинг 5.3. Генерація трикутного сигналу за допомогою R-2R ЦАП.

```
/*
 * R_2R.c
 * Генерація трикутного сигналу за домомогою R-2R ЦАП
 * Із фіксованими інтервалами часу на ЦАП подається байт значення якого монотонно зростає
від 0 до 255 після цього значення монотонно спадає від 255 до 0. Проміжки часу
відраховуються за допомогою переривання, по співпадінню TCNT0  регістрів OCMR0A таймера TC0
 */
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
```

```c
#include <util/delay.h>

/* порт для R-2R ЦАП */
#define R2R_Port PORTD
#define R2R_DDR     DDRD
#define dt 5  // інтервал часу запису даних в ЦАП

unsigned char Data = 0; // значення, яке буде подаватися на ЦАП
unsigned char Dir_F=1; //прапорець, який вказує на напрямок 1: Data 0 -> 255, 0: Data 255->0

/* прототипи функцій */
void Init_TC0(void); // ініціалізація таймера TC0 режим роботи CTC
void Init_Port(void); // ініціалізація порту для ЦАП

void Init_Port(void)
{
	R2R_DDR = 0xff; // весь порт на вихід
	R2R_Port = 0;
}

void Init_TC0(void)
{
	TCNT0 = 0;
	OCR0A = dt;
	TCCR0A |= (1<<WGM01); // режим CTC
	TIMSK0 |=(1<<OCIE0A); // дозволяємо переривання по співпадінні значень OCR0A та TCNT0
	TIMSK0 |=(1<<OCF0A); // скидаємо прапорець переривання
	TCCR0B |= (1<<CS00); // вмикаємо таймер, частота роботи TC0 рівна F_CPU
}
ISR(TIMER0_COMPA_vect)
{
	R2R_Port = Data; // записуємо дані призначені для ЦАП
	if (Dir_F == 1) // якщо напрямок 0->255 пряма лічба
	{
		Data+=5; // збільшуємо Data
		if (Data == 255) // перевіряємо чи дійшли до вершини трикутника
		{
		  Dir_F = 0; // міняємо напрямок
		  Data = 250; // починаємо зменшувати значення Data
		}
	}
	if ( Dir_F == 0) // якщо напрямок 255->0 зворотна лічба
	{
		Data-=5; // зменшуємо Data
		if (Data == 0) // якщо опустилися до основи, то міняємо напрямок
		{
		 Dir_F = 1;
		 Data = 5; // збільшили Data на 5
		}
	}

	TIMSK0 |=(1<<OCF0A); // очищаємо прапорець перривання
}
int main(void)
{
	Init_Port();
	Init_TC0();
	sei();

	DDRC=0xff;

    while (1)
    {
		PINC |= (1<<PC0); // фонова задача
		_delay_ms(150);
    }
}
```

Результат виконання програми можна спостерігати на осцилографі (рис. 6)

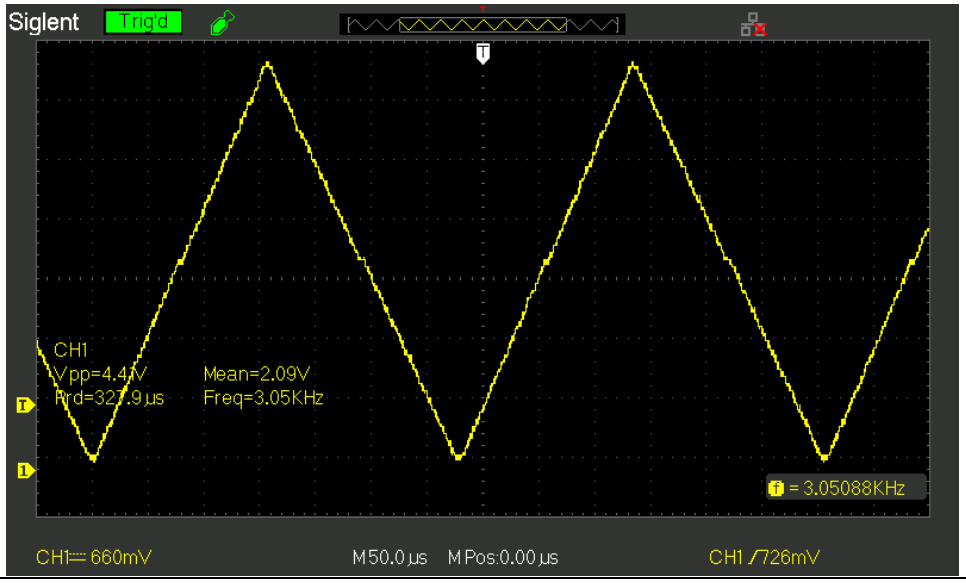


Рис. 6. Генерація трикутного сигналу за допомогою ЦАП R-2R матриці.

Завдання до лабораторної роботи.

1. Створити проект в Atmel Studio 7, надрукувати програму наведену в лістинг .5.2 Вивчити логіку роботи програми за допомогою симулятора.
2. Створити проект в Atmel Studio 7, надрукувати програму наведену в лістинг 5.3. Вивчити логіку роботи програми за допомогою симулятора.

Розробити алгоритм, написати програму, відладити роботу за допомогою симулятора та перевірити за допомогою лабораторного стенду:

3. «Багато-канальний вольтметр». Пристрій, який проводить вимірювання напруги в діапазоні 0-1,1 В із точністю 1 мВ від різних джерел (використати вхідні 4 канали АЦП: ADC7, ADC6, ADC5, ADC4). Вимірювання проводити із періодом в 1 мс, усереднювати по 32 вимірюванням і результат виводити на 7 сегментний дисплей. Передбачити такі режими переключення вхідних каналів АЦП: 1) циклічний в кожному каналі вимірювання проводити протягом 1 с – результат усереднювати та виводити на 7 сег.дисплей; 2) послідовне переключення кнопкою «Канал»: при натисканні кнопки перемикається канали ADC7→ADC6→ADC5→ ADC4→ ADC7→ …, при включенні відповідного каналу засвічується відповідний йому світлодіод. Режими переключати за допомогою кнопки «Режим». Передбачити можливість передачі результатів вимірювання через USART у вигляді строки: «#k 923 mV \n», де #k – номер каналу, 923 mV – значення напруги, \n – «new line» символ. Передача включається і виключається за допомогою кнопки «Передача», або за запитом від ПК – запит повинен містити номер каналу.
4. Вивчення паралельного R-2R ЦАП. Написати програму, яка послідовно видає паралельний 8-bits код на R2R ЦАП і одночасно вимірює значення напруги на виході ЦАП за допомогою вбудованого АЦП. Опорна напруга – напруга живлення. Дані усереднювати по 16-ти значенням і передавати у вигляді строки «123 456 \n», де 123 - код на вході ЦАП, 456 – код АЦП. Передавати по 8 рядків за командою від ПК. За отриманими результатами побудувати графік «напруга на виході ЦАП – код ЦАП» Визначити характеристики ЦАП: адитивна та мультиплікативна похибки, диференціальна та інтегральна нелінійність.

Контрольні запитання:

1. Поясніть принцип аналогово-цифрового перетворення.
2. Роздільна здатність та похибки аналогово-цифрового перетворення.
3. Характеристики вбудованого АЦП в ATmega328P.
4. Режими роботи вбудованого АЦП в ATmega328P.
5. Контрольні та робочі регістри вбудованого АЦП в ATmega328P.
6. Методика налаштування роботи вбудованого АЦП в ATmega328P.
7. Принципи цифро-аналогового перетворення: ШІМ, ЦАП на основі вагових резисторів, R-2R матриця.

Література

1. Datasheet ATmega328P. http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf
2. Программирование микроконтроллеров ATMEL на языке С. Прокопенко В.С., К.: «МК-Пресс», 2012. – 320с.
3. Програмирование на языке С для AVR и PIC микроконтролеров. Шпак Ю.А., К.: «МК-Пресс», 2016. – 544стр.
4. Микроконтроллеры AVR семейств Tiny и Mega фирмы ATMEL. – М.: Издательский дом «Додэка-XXI», 2004. – 560 с.
5. Application Note_1497 AVR035: Efficient C Coding for 8-bit AVR microcontrollers. http://ww1.microchip.com/downloads/en/AppNotes/doc1497.pdf
6. Application Note_AVR42787 AVR Software User Guide http://ww1.microchip.com/downloads/en/AppNotes/Atmel-42787-AVR-Software-User-Guide_ApplicationNote_AVR42787.pdf

Лабораторна робота 6.

Використання вбудованого модуля SPI.

Мета: навчитися використовувати вбудований в ATmega328P модуль SPI для взаємодії із зовнішніми пристроями.

Для обміну даними між мікроконтролером та зовнішніми периферійними пристроями (модулі та карти пам'яті, LCD дисплеї, АЦП, ЦАП, цифрові потенціометри, тощо) або іншими мікроконтролерами, використовується швидкісний, синхронний, послідовний інтерфейс, який називають SPI (Serial Peripheral Interface). Мікроконтролери AVR серії ATmega мають вбудований апаратний модуль, який забезпечує обмін даними за цим протоколом. В процесі обміну пристрої діляться на «ведучий» (Master) та «ведений» (Slave). Для обміну використовуються 4 лінії:

- **SCK** - Тактовий сигнал. Використовується для синхронізації даних (на цьому виводі master генерує синхроімпульси);
- **MOSI** (Master Out , Slave In) - передавач ведучого , приймач веденого (за цим виводом master передає дані slave);
- **MISO** (Master In , Slave Out) - приймач ведучого , передавач веденого (за цим виводом master приймає дані від slave);
- $\overline{SS}$ (Slave Select) - вибір веденого.

Як правило, МК виступає в ролі ведучого, ведучий пристрій, за допомогою сигналу $\overline{SS}$, переводить ведений пристрій в активний режим, формує тактовий сигнал **SCK** та передає дані через **MOSI.** Одночасно ведений пристрій передає свої дані через **MISO.** Модуль SPI апаратно реалізується за допомогою пари регістрів зсуву (рис. 1).

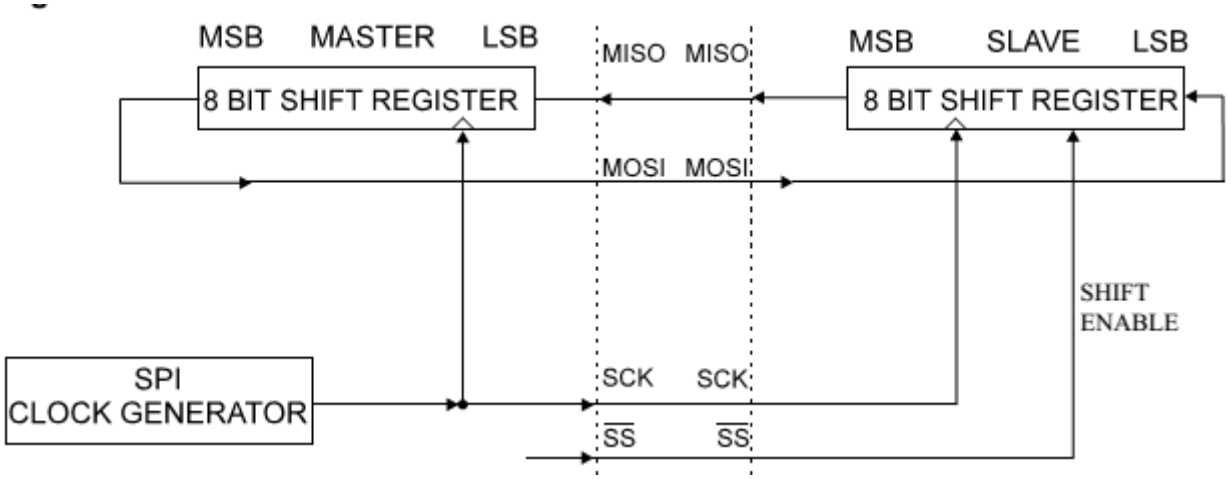


Рис. 1. З'єднання SPI між ведучим та веденим пристроєм.

Передачу даних завжди ініціює ведучий пристрій, для цього на лінію $\overline{SS}$ встановлюється низький рівень. Дані, які призначені для передачі, повинні бути поміщені в регістри зсуву ведучого та веденого пристрою. Коли на лінії синхронізації з'являється фронт сигналу (передній чи задній – це визначається налаштуваннями модуля SPI), регістри зсуву у ведучого та веденого пристрою обміняються одним бітом. Вміст регістрів зміститься ліворуч і старші біти кожного із регістрів перейдуть в молодші біти регістрів з'єднаних пристроїв – ведучий та ведений обміняються бітами. Іншими словами, ведений передасть свій старший біт через лінію MISO ведучому, який запише його в розряд, який звільнився (за рахунок зсуву через лінію MOSI в молодший розряд веденого). Протягом 8-ми тактових імпульсів, регістри ведучого та веденого пристрою обміняються вмістом. Таким чином, прийом та передача між пристроями відбувається одночасно – реалізується дуплексний режим передачі даних. Після закінчення обміну даними, ведучий пристрій встановлює на лінії $\overline{SS}$ високий рівень напруги.

До одного МК, який працює ведучим, можна підключати декілька ведених пристроїв через SPI, для цього лінії **SCK**, **MOSI**, **MISO** включаються паралельно, а лінія $\overline{SS}$ обирається для кожного веденого пристрою окремо (рис. 2). В кожен момент часу обмін даними відбувається лише із одним веденим пристроєм, а решта є не активними. Наприклад : для передачі даних до пристрою №1 (рис.2) ведучий пристрій переводить стан його $\overline{SS}_1$ лінії на низький рівень, а на лінію $\overline{SS}_2$ встановлюється високий рівень напруги.

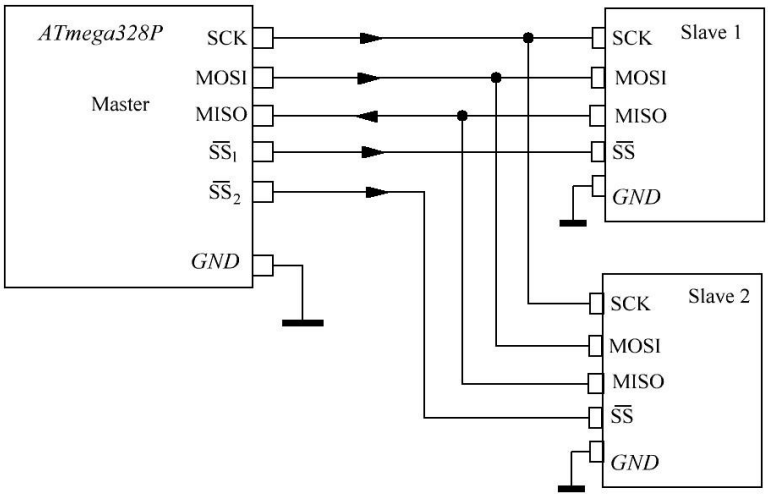


Рис. 2. З'єднання ATmega328P, в якості ведучого пристрою, для обміну даними із двома веденими пристроями.

Основні характеристики модуля SPI МК ATmega328P (детальний опис модуля SPI наведений в технічній документації розділ 23, SPI ст.215 – 225):

- дуплексний режим синхронної передачі даних;
- ведучий (Master), або ведений (Slave) режим роботи;
- сім програмованих швидкостей передачі даних;
- генерація переривання «закінчення передачі даних» (SPI STC – SPI Serial Transfer Complete);

В ATmega328P модуль SPI використовує 4 лінії, які суміщені із лініями PORTB: SCK – PB5, MOSI – PB3, MISO – PB4, $\overline{SS}$ - PB2. Якщо модуль SPI в ATmega328P працює як ведучий, то лінія $\overline{SS}$ не контролюється автоматично, управління нею повинно відбуватися програмно.

Протокол обміну по SPI.

Специфікація SPI передбачає чотири режими обміну даними, які відрізняються між собою співвідношенням фази і полярності тактового сигналу SCK по відношенню до сигналу даних. Ці режими встановлюються двома бітами в регістрі SPCR0:

CPOL (Clock Polarity) – полярність тактового сигналу SCK, визначає початковий рівень напруги (низький/високий) сигналу SCK;

CPHA (Clock Phase) – фаза тактового сигналу, визначає послідовність установки та вибірки даних.

Режими обміну даними в залежності від комбінації бітів CPOL, CPHA:

SPI режим 0. CPOL=0, CPHA=0. Тактовий сигнал починається із рівня логічного нуля, вибірка даних відбувається по передньому фронту SLK, а встановлення наступного біта відбувається по задньому фронту SCK (рис. 3, а).

SPI режим 1. CPOL=0, CPHA=1. Тактовий сигнал починається із рівня логічного нуля. Установка наступного біту відбувається по передньому фронту SCK, вибірка біта – по задньому фронту SCK (рис.3,б).

SPI режим 2. CPOL=1, CPHA=0. Тактовий сигнал починається із рівня логічної одиниці, вибірка біта відбувається по задньому фронту SCK. Установка наступного біту виконується по передньому фронту SCK (рис.3, в).

SPI режим 3. CPOL=1, CPHA=1. Тактовий сигнал починається із рівня логічної одиниці, вибірка біта відбувається по передньому фронту SCK. Установка наступного біту виконується по задньому фронту SCK (рис.3, г).

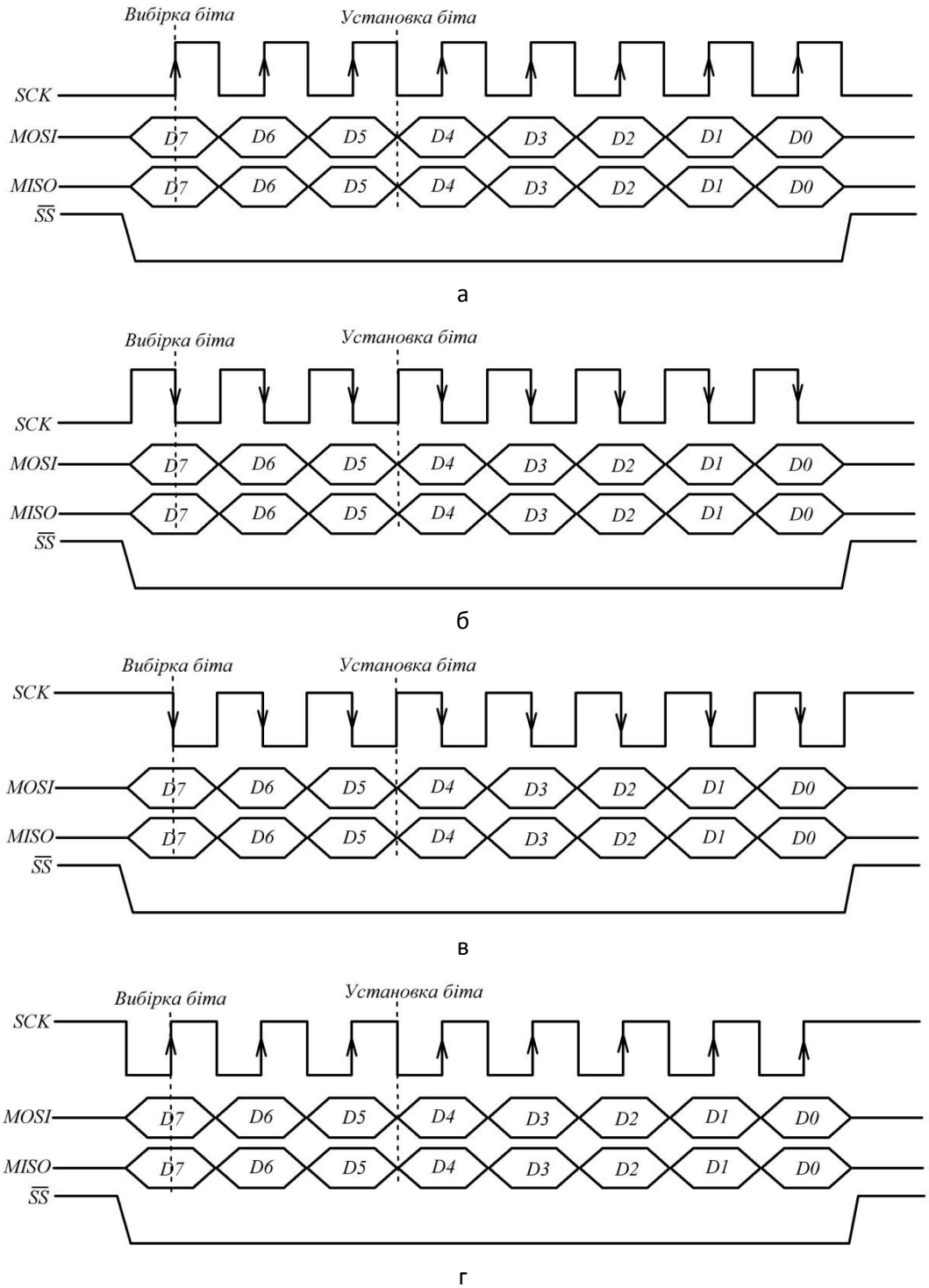


Рис. 3. Режими роботи SPI: а) SPI режим 0; б) SPI режим 1; в) SPI режим 2; г) SPI режим 3.

Опис регістрів модуля SPI.

Для налаштування режиму роботи та обміну даними модуль SPI має 3 регістри:

SPCR (SPI Control Register) – контрольний регістр;

SPSR (SPI Status Register) – регістр статусу;

SPDR (SPI Data Register) – регістр даних.

SPCR (SPI Control Register).

bit	7	6	5	4	3	2	1	0
SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

SPIE (SPI Interrupt Enable) – якщо цей біт встановлений, то по закінченню обміну даними, буде сформований запит на переривання, якщо переривання дозволені глобально.

SPE (SPI Enable) – якщо біт SPE0 встановлений, то модуль SPI буде включений. Біт SPE0 потрібно встановлювати перед виконанням будь-яких операцій із модулем SPI.

DORD (Data Order) – цей біт встановлює порядок передачі даних: якщо DORD0=1, то першим буде передаватися наймолодший біт (LSB). Якщо DORD=0, то першим буде передаватися найбільш значущий біт (MSB).

MSTR (Master/Slave Select). Якщо MSTR=1 то SPI модуль буде в режимі ведучого пристрою (Master), а якщо MSTR=0, то SPI модуль буде в режимі веденого пристрою (Slave). Якщо SPI модуль в режимі веденого (MSTR=0), то лінія $\overline{SS}$ завжди є входом, якщо на $\overline{SS}$ присутній низький рівень напруги, модуль SPI вмикається і готовий приймати вхідні дані. Якщо на лінії $\overline{SS}$ високий рівень напруги, то модуль SPI є не активним і не буде приймати вхідні дані.

Якщо модуль SPI в режимі ведучого пристрою (MSTR=1), то напрям роботи лінії $\overline{SS}$ встановлюється програмно, якщо $\overline{SS}$ вихід, то сигнали на цій лінії не впливають на роботу SPI модуля. Якщо лінія $\overline{SS}$ сконфігурована на вхід, то на ній повинен бути присутній високий рівень напруги, для коректної роботи SPI модуля в режимі ведучого пристрою. Якщо зовнішній пристрій встановлює на лінії $\overline{SS}$ низький рівень, то SPI модуль переходить в режим веденого пристрою.

CPOL (Clock Polarity) –визначає початковий рівень напруги (низький/високий) сигналу SCK, якщо CPOL=1, то початковий рівень напруги на лінії SCK високий (рис.3 в, г), а за умови CPOL=0 – низький (рис.3 а, б).

CPHA (Clock Phase) – фаза тактового сигналу, визначає послідовність установки та вибірки даних. Якщо CPHA=1, то дані встановлюються за першим фронтом сигналу SCK, а зчитуються за другим фронтом (рис. 3 б, г), а якщо CPHA=0, то навпаки: вибірка даних за першим фронтом, а установка за другим фронтом сигналу SCK (рис. 3 а, в).

SPR1, SPR0 (SPI Clock Rate Select) – ці біти, разом із бітом SP2X регістру SPSR , встановлюють значення тактової частоти SCK. Залежність тактової частоти від конфігурації цих бітів наведено в таблиці 1.

Таблиця 1. Залежність тактової частоти модуля SPI від системної частоти процесора ***F_CPU***

SP2X	SPR1	SP00	Частота SCK
0	0	0	***F_CPU/4***
0	0	1	***F_CPU/16***
0	1	0	***F_CPU/64***
0	1	1	***F_CPU/128***
1	0	0	***F_CPU/2***
1	0	1	***F_CPU/8***
1	1	0	***F_CPU/32***
1	1	1	***F_CPU/64***

SPSR (SPI Status Register).

bit	7	6	5	4	3	2	1	0
SPSR	SPIF	WCOL	-	-	-	-	-	SPI2X

SPIF (SPI Interrupt Flag) – по закінченню обміну даними через модуль SPI , встановлюється SPIF. Якщо дозволене переривання по закінченню передачі і дозволені переривання глобально, то буде згенерований запит на переривання. Після закінчення виконання функції

обробника переривання, SPIF скидається автоматично, якщо переривання заборонені, то SPIF скидається при зверненні до регістру даних SPDR.

WCOL (Write Collision Flag) – цей біт встановлюється, якщо записати дані в регістр даних SPDR, під час процесу обміну даними. Цей біт буде скинутий автоматично після програмного звернення до регістрів SPSR, SPDR.

SPI2X (Double SPI Speed) – якщо цей біт встановлений, то частота тактового сигналу буде подвоєна (таблиця 1).

SPDR (SPI Data Register)

Bit	7	6	5	4	3	2	1	0
SPDR	SPID7	SPID6	SPID5	SPID4	SPID6	SPID2	SPID1	SPID0

SPID[7..0] SPI Data. Регістр **SPDR** використовується для обміну даними між регістром зсуву та оперативною пам'яттю мікроконтролера. Запис байту в цей регістр починає процес передачі даних. Отримані дані можна прочитати із цього ж регістру.

Функції ініціалізації модуля SPI та обміну даними

Початкове налаштування модуля SPI зводиться до вибору режиму роботи ведучий/ведений, встановлення частоти SCK, налаштування портів. Приклад функції налаштування наведений в лістинг 6.1.

Лістинг 6.1. Налаштування модуля SPI в режимі ведучий пристрі.

```
/* Порти для SPI */
#define DDR_SPI DDRB
#define SPI_PORT PORTB
#define MOSI PB3
#define MISO PB4
#define SS PB2
#define SCK   PB5

void SPI_Master_Init(void)
{
    DDR_SPI |= (1<<MOSI)|(1<<SCK)|(1<<SS); //MOSI, SCK, SS - вихід
    DDR_SPI &= ~(1<<MISO); // MISO - вхід
    /*вмикаємо модуль SPI, режим Master, частота F_CPU/16 = 1 МГц режим SPI 0*/
    SPSR |= (1<<SPE)|(1<<MSTR)|(1<<SPR0);
}
```

Прийом та передача даних

Процес передачі та прийому даних, за допомогою модуля SPI, який працює в режимі ведучого пристрою (Master) складається із такої послідовності дій:

1. встановити низький рівень на лінії $\overline{SS}$;
2. завантажити байт даних в регістр SPDR;
3. очікувати закінчення передачі – перевірка прапорця SPIF;
4. прочитати значення регістру SPDR і зберегти байт за потреби;
5. якщо потрібно передати декілька байтів, то повернутися на крок 2;
6. після передачі/прийому останнього байту встановити високий рівень напруги на лінії $\overline{SS}$.

Приклад програмного коду для прийому та передачі по SPI наведений в лістингу 6.2.

Лістинг 6.2. Функції прийому/передачі даних по SPI.

```
unsigned char SPI_Data_Transfer(unsigned char Data_Out)
{
    unsigned char Data_In=0;
    SPI_PORT &= (1<<SS); // лінію SS=0
       SPDR = Data_Out; // починаємо передавати Data_Out
       while(SPSR & (1<<SPIF)); // очікуємо закінчення передачі
       Data_In = SPDR; // те, що отримали записуємо в пам'ять
       return Data_In;
}
```

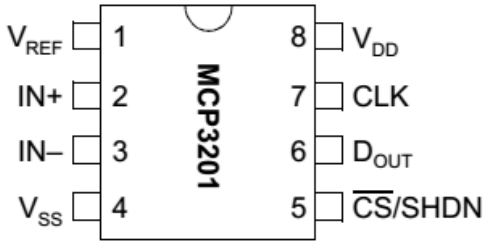
Використання модуля SPI.

Використання модуля SPI, вбудованого в ATmega328P, розглянемо на прикладі використання зовнішнього пристрою MCP3201.

MCP3201 – це 12 бітний аналогово-цифровий перетворювач, який має послідовний інтерфейс обміну даними сумісний із SPI (режим 0 або 1). АЦП MCP3201 має точність не меншу ±1 LBS, максимальну швидкість перетворення 100 kSPS (за напруги живлення +5 В). Більш детальна інформація міститься в технічному описі на сайті виробника:

http://www.microchip.com/wwwproducts/en/MCP3201

Мікросхема MC3201 поміщена в корпус, який має 8 виводів. Позначення та функціональне призначення виводів наведено на рис. 4.



Позначення	Функція
V_{DD}	Наруга живлення: +2,7 - +5 В
V_{SS}	GND – земля
$IN+$ $IN-$	аналоговий вхід
CLK	Тактова частота для послідовної передачі даних
D_{OUT}	Дані
$\overline{CS}/SHDN$	Вибір мікросхеми / сигнал виключення
V_{REF}	вхід для опорної напруги

Рис. 4. Розташування та призначення виводів MC3201.

Схема підключення MC3201 до МК ATmega328P, джерела опорної напруги та живлення наведена на рис. 5.

Вхідна аналогова напруга V_{IN}, через вхідний повторювач напруги LMV321, подається на входи $IN+$, $IN-$ АЦП, в якості джерела опорної напруги використана мікросхема MC1541, виходи послідовного інтерфейсу *CLK*, D_{OUT}, $\overline{CS}/SHDN$ під'єднані до ліній модуля SPI в ATmega328P: *SCK, MISO,* $\overline{SS}$ відповідно.

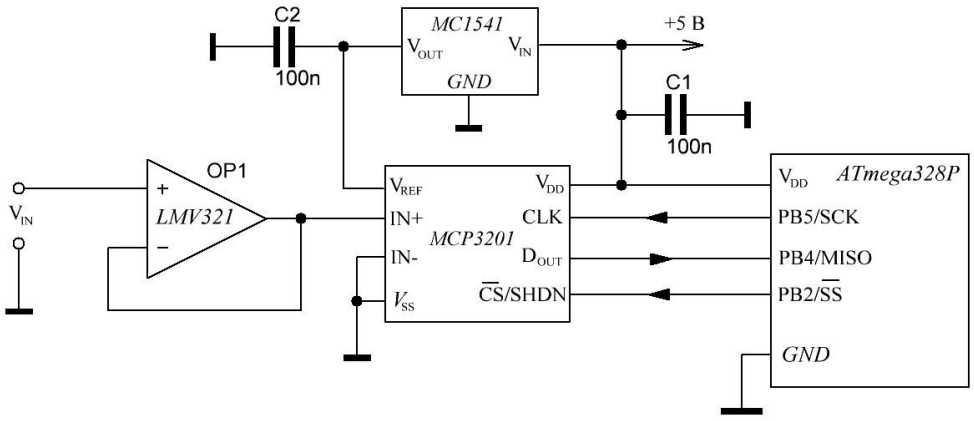


Рис. 5. Схема підключення АЦП MCP3201 до МК ATmega328P.

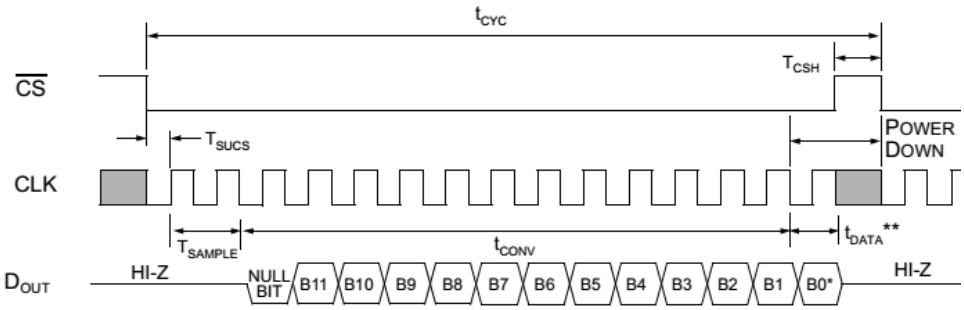
Значення вихідного коду АЦП D_{OUT} буде залежати від вхідної та опорної напруги:

$$D_{OUT} = \frac{2^{12}}{V_{REF}} V_{IN} = \frac{4096}{V_{REF}} V_{IN}, \qquad (1)$$

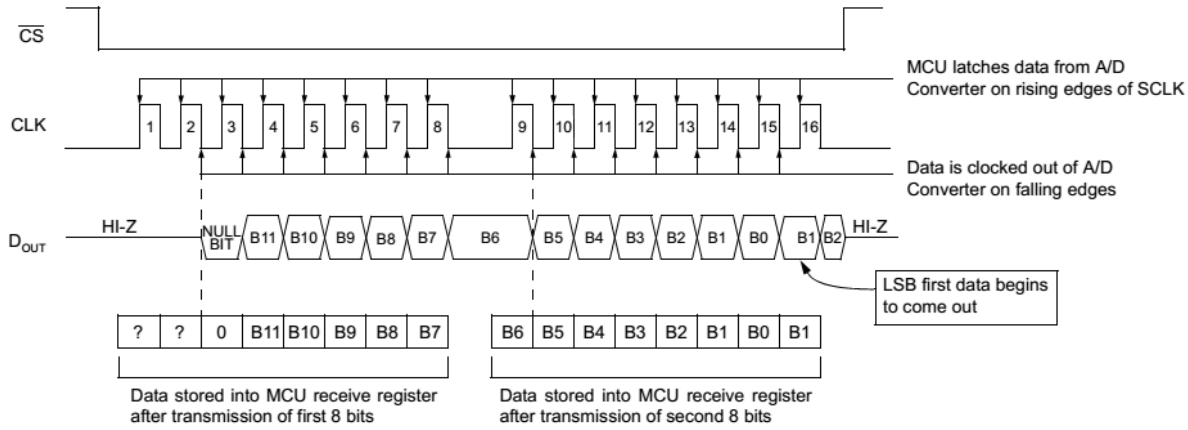
де $V_{IN} = V(IN+) - V(IN-)$ різниця потенціалів на аналоговому вході АЦП.

Джерело опорної напруги MC1541 забезпечує стабільну напругу $V_{REF} = 4096\,мВ$, тому відповідно до (1), вихідний код АЦП буде відповідати вхідній напрузі вираженій в мВ.

Аналогово-цифрове перетворення починається, коли на лінії $\overline{CS}/SHDN$ буде встановлений низький рівень напруги. Із першим переднім фронтом *CLK* почнеться перетворення, яке триватиме 15 тактових імпульсів. Починаючи із 3-го на виході D_{OUT} будуть з'являтися дані (рис. 6, а). Оскільки, за протоколом, SPI за один цикл передається 8 біт, то для взаємодії із 12-ти розрядним АЦП потрібно виконати обмін даними між ATmega328P та MC3201 два рази (рис. 6, б), та провести маніпуляції із отриманими байтами, щоб отримати правильний результат.



а



б

Рис. 6. Часові діаграми обміну даними між ATmega328P та MC3201.

Програма, яка реалізує обмін даними між МК та MCP3201, наведена в лістингу 6.3. Функції передачі даних через USART використані із лабораторної роботи №4 (лістинги 4.1 – 4.3), але винесені в окремі файли `usart.h, usart.c`.

Лістинг 6.3. Використання модуля SPI для обміну даними із АЦП MCP3201.

```
/*
 * lab6-1.c
 *Використання модуля SPI
 * на прикладі обміну даними із АЦП MCP3201
 * виміряні значення відсилаються  через USART. у вигляді ASCII строки
 */
#define F_CPU 16000000UL
#define ubbr 51
#include <avr/io.h>
#include <util/delay.h>
#include "usart.h"

/* Порти для SPI */
```

```
#define DDR_SPI DDRB
#define SPI_PORT PORTB
#define MOSI PB3
#define MISO PB4
#define SS PB2
#define SCK   PB5

unsigned char Number_String[4]={0x30};
unsigned char Ready[]="Ready\n";
unsigned char V[]="\n V=";
unsigned char mV[]=" mV";
unsigned char Data[]="";

void SPI_Master_Init(void);
unsigned int MCP3201_Get_Data(void);
void Int_To_Array( unsigned char* Number_String, int Data);

void SPI_Master_Init(void)
{
    DDR_SPI |= (1<<MOSI)|(1<<SCK)|(1<<SS); //MOSI, SCK, SS - вихід
    DDR_SPI &= ~(1<<MISO); // MISO - вхід
       SPI_PORT |=(1<<SS); //  на лінію SS встановити високий рівень напруги
       /*вмикаємо модуль SPI, режим Master, частота F_CPU/16 = 1 МГц, режим 0*/
    SPCR |= (1<<SPE)|(1<<MSTR)|(1<<SPR0);
}
/* функція, яка перетворює число в масив ASCII символів */
void Int_To_Array( unsigned char* Number_String, int Data)
{
       /* обнуляємо попереднє значення - записуємо символ "0" */
       Number_String[0]=Number_String[1]=Number_String[2]=Number_String[3]=0x30;
       while (Data>=1000)              // вичисляємо кількість тисяч
       { Data-=1000; Number_String[3]++; }
       while (Data>=100)             // вичисляємо кількість сотень
       { Data-=100; Number_String[2]++; }
       while (Data>=10)                          // вичисляємо кількість десятків
       { Data-=10; Number_String[1]++; }
       Number_String[0]+=Data;               // те що лишилося - кількість одиниць
       /* якщо старші розряди 0, то їх не відображаємо, замість символів "0" відішлемо ' '
*/
       if(Number_String[3]==0x30) Number_String[3]=0x20;
       {
       if (Number_String[3]==0x20 && Number_String[2]==0x30) {Number_String[2]=0x20;}
       if (Number_String[3]==0x20 && Number_String[2]==0x20 && Number_String[1]==0x30)
              { Number_String[1]=0x20;}
       }
       }
/* функція, яка зчитує дані із  АЦП MCP3201 через SPI*/
unsigned int MCP3201_Get_Data(void)
{
    unsigned char Hi_B=0; // старший біт даних
    unsigned char Low_B=0; // молодший біт
       unsigned int Data_Out=0;
       SPI_PORT &= ~(1<<SS); // лінію SS=0 - вмикаємо MCP3201
       /* починаємо обмін даними */
       SPDR = 0x00; // регістр даних SPI записуємо довільне число
       while( !(SPSR & (1<<SPIF)) ); // очікуємо закінчення передачі
    Hi_B = SPDR; // оптимали старший байт
    SPDR = 0x00;
       while( !(SPSR & (1<<SPIF)) );
    Low_B = SPDR; // оптимали молодший байт
       SPI_PORT |= (1<<SS); // лінію SS=1 - вимикаємо MCP3201
       /* маніпуляції із байтами */
       Data_Out = ( (Hi_B<<8)|Low_B )>>1; // об"єднуємо старший та молодший біт в int
       Data_Out &=0x0FFF; // накладаємо маску щоб виділити 12 молодших біт
       return Data_Out; //
}
```

```
int main(void)
{
	SPI_Master_Init();
	USART_Init(ubbr);
	USART_Send_Str(Ready);
	unsigned int D=0;
    while (1)
    {
		USART_Send_Str(V);
		D = MCP3201_Get_Data();
		Int_To_Array(Number_String, D); // число перетворили в масив ASCII символів
		/* відсилаємо масив */
		unsigned char i=3;
		do
		{
			USART_Send_Byte(Number_String[i]);
		} while (i--);
		USART_Send_Str(mV);
		_delay_ms(250);
    }
}
```

Функція `MCP3201_Get_Data()` зчитує дані із АЦП MCP3201, логіка її роботи відповідає діаграмі на рис. 6, б. На лінії $\overline{SS}$ встановлюється низький рівень напруги, після цього 2 рази проводиться обмін даними, кожного разу значення регістру SPDR зберігається в змінних Hi_B, Low_B і тільки після цього, на лінію $\overline{SS}$ встановлюється високий рівень напруги. Для того, щоб отримати правильний результат аналогово-цифрового перетворення, потрібно провести маніпуляції із байтами Hi_B, Low_B:

- об'єднати старший Hi_B і молодший Low_B байти в одну 16-ти розрядну заміну і зсунути отриманий результат праворуч на один розряд:

  `Data_Out = ( (Hi_B<<8)|Low_B )>>1;`
- оскільки, 2 старші біти Hi_B є невизначеними, то потрібно із отриманого числа `Data_Out` виділити молодші 12 розрядів. Це найпростіше виконати за допомогою порозрядного І із величиною 0x0FFF=0b111111111111:

  `Data_Out &=0x0FFF;.`

Значення `Data_Out` повертається функцією `MCP3201_Get_Data()`.

Завдання до лабораторної роботи.

1. Створити проект в Atmel Studio 7, надрукувати програму наведену в лістингу 6.3 Вивчити логіку роботи програми за допомогою симулятора.

Розробити алгоритм, написати програму, відладити роботу за допомогою симулятора та перевірити за допомогою лабораторного стенду:

2. Цифровий осцилограф (реєстратор форми сигналу). Модифікувати програму наведену в лістингу 6.3 – перетворити в пристрій для дослідження форми сигналу (цифровий осцилограф). Алгоритм роботи: вхідна напруга V_{IN} постійно вимірюється за допомогою АЦП MCP3201 із інтервалом в 25 мкс. При настанні певної події АЦП проводить 750 разів вимірювання напруги вхідного сигналу із максимально можливою швидкістю. Виміряні значення поміщаються в масив. По закінченні вимірювання, кожне значення напруги перетворюється в символьний рядок ASCII символів і пересилається на ПК.
   Події, які запускають аналогово-цифрове перетворення:
   - команда від ПК передана через USART;
   - вхідна напруга перевищує заданий рівень сигналу V_{TRIG} (значення V_{TRIG} задається від ПК через USART), наприклад: якщо $V_{IN} > 50\ мВ$, то запускається процес запису;
   - натискання кнопки.
   Використовуючи генератор сигналів в якості джерела вхідної напруги, визначити:
   - максимально можливу частоту дискретизації цифрового осцилографа;

- максимально можливу частоту, яка буде відтворюватися без спотворень.

Контрольні запитання:

1. Принцип обміну даними за допомогою SPI інтерфейсу, електричне з'єднання пристроїв, поділ на ведучі та ведені пристрої.
2. Режими обміну даними.
3. Контрольні та статусні регістри модуля SPI в ATmega328P.
4. Методика налаштування роботи вбудованого SPI в ATmega328P в режимі ведучий пристрій.
5. Прийом та передача 1 байта за допомогою SPI.
6. АЦП MCP3201, особливості обміну даними за протоколом SPI

Література

1. Datasheet ATmega328P.
2. Программирование микроконтроллеров ATMEL на языке С. Прокопенко В.С., К.: «МК-Пресс», 2012. – 320с.
3. Програмирование на языке С для AVR и PIC микроконтролеров. Шпак Ю.А., К.: «МК-Пресс», 2016. – 544стр.
4. Микроконтроллеры AVR семейств Tiny и Mega фирмы ATMEL. – М.: Издательский дом «Додэка-XXI», 2004. – 560 с.
5. Технічний опис MCP3201 http://www.microchip.com/wwwproducts/en/MCP3201
6. Технічний опис MCP1541 http://www.microchip.com/wwwproducts/en/mcp1541
7. Application Note_1497 AVR035: Efficient C Coding for 8-bit AVR microcontrollers. http://ww1.microchip.com/downloads/en/AppNotes/doc1497.pdf
8. Application Note_AVR42787 AVR Software User Guide http://ww1.microchip.com/downloads/en/AppNotes/Atmel-42787-AVR-Software-User-Guide_ApplicationNote_AVR42787.pdf