МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ
ІМЕНІ АДМІРАЛА МАКАРОВА

КАФЕДРА ПРОГРАМОВАНОЇ ЕЛЕКТРОНІКИ, ЕЛЕКТРОТЕХНІКИ І ТЕЛЕКОМУНІКАЦІЙ

РЯБЕНЬКИЙ В.М., УШКАРЕНКО О.О.

МЕТОДИЧНІ ВКАЗІВКИ
ДО ВИКОНАННЯ ПРАКТИЧНИХ РОБІТ
з дисципліни
«ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ ВБУДОВАНИХ МІКРОПРОЦЕСОРНИХ СИСТЕМ»

Рекомендовано Навчально-методичною радою НУК

Електронне видання

Миколаїв • НУК • 2024

УДК 621.38:004(076)
Р 98

Автори: **Рябенький Володимир Михайлович,** докт. техн. наук, професор, завідувач кафедри програмованої електроніки, електротехніки і телекомунікацій Національного університету кораблебудування імені адмірала Макарова;
Ушкаренко Олександр Олегович, докт. техн. наук, професор, професор кафедри програмованої електроніки, електротехніки і телекомунікацій Національного університету кораблебудування імені адмірала Макарова.

Рецензенти: **Черно Олександр Олександрович**, докт. техн. наук, професор, завідувач кафедри КСУ НУК ім. адм. Макарова;
Дьяконов Олексій Сергійович, канд. техн. наук, доцент, доцент кафедри ПЕЕТ НУК ім. адм. Макарова

Рекомендовано Навчально-методичною радою Національного університету кораблебудування імені адмірала Макарова (протокол № 1 від 21 лютого 2024 р).

Рябенький В.М.

Р98 Методичні вказівки до виконання практичних робіт з дисципліни “Об’єктно-орієнтоване програмування вбудованих мікропроцесорних систем” / В.М. Рябенький, О.О. Ушкаренко. – Миколаїв : НУК, 2024. – 175 с.

Методичні вказівки до виконання практичних робіт присвячені опису принципів проектування архітектури програмного забезпечення для вбудованих мікропроцесорних систем та розробки драйверів периферійних вузлів мікроконтролера на мові програмування С++, особливостей використання шаблонів проектування при вирішенні різноманітних завдань збору, обробки та відображення інформації. На великій кількості прикладів продемонстровано принципи створення класів, що надають високорівневий інтерфейс для взаємодії з апаратними засобами, організацію взаємодії між класами з використанням відношень композиції та/або асоціації. З метою отримання практичних навичок не лише з розробки програмного забезпечення, але й проектування вбудованих мікропроцесорних систем, в кожній практичній роботі приводиться принципова схема досліджуваної системи.

Практичні роботи проводяться з використанням спеціалізованих стендів або в середовищі моделювання, в якому виконується розробка схемотехнічних рішень та моделювання роботи вбудованих мікропроцесорних систем, що мають в своєму складі засоби відображення інформації (світлодіоди, семисегментні індикатори, рідкокристалічні індикатори), різноманітні датчики фізичних величин (температури, освітленості, концентрації газів, відстані тощо), виконавчі механізми у вигляді двигунів постіного струму, крокових двигунів або сервоприводів, та комунікаційні інтерфейси (RS-232, SPI, I^2C, 1-Wire).

Призначено для студентів спеціальностей 171 «Електроніка» та 172 «Електронні комунікації та радіотехніка».

УДК 621.38:004(076)

ЗМІСТ

Практична робота 1.

Тема: Знайомство з середовищем розробки Microchip Studio 7, принципами створення та налаштування проекту. Розробка драйвера портів мікроконтролера ATMega16.

Середовище розробки програмного забезпечення Microchip Studio дозволяє створювати програми для вбудованих мікропроцесорних систем як на асемблері, так і на мовах C/C++. Середовище розробки містить майстер проектів, віртуальний симулятор, редактор вихідного коду, модуль внутрішньосхемного налагодження та інтерфейс командного рядка. Підтримуються компілятор GCC та плагін AVR RTOS (операційної системи реального часу). Розробники можуть вибрати найбільш оптимальні для проекту способи кодування. Візуальні інструменти дають змогу прискорити написання програми. Завдяки зв'язці програмних пакетів Microchip Studio (попередня назва – Atmel Studio) і Proteus від фірми Labcenter Electronics можливе програмування мікроконтролерів без будь-якої матеріальної бази. Microchip Studio по праву вважається найкращим середовищем створення програм для мікроконтролерів AVR. Середовище розробки програмного забезпечення Microchip Studio 7 для мікроконтролерів можна завантажити за посиланням:

https://www.microchip.com/en-us/tools-resources/develop/microchip-studio

Процес установки середовища розробки є стандартним для Windows. Після установки середовища розробки Microchip Studio 7 та його запуску відкриється головне вікно, яке представлено на рис. 1.

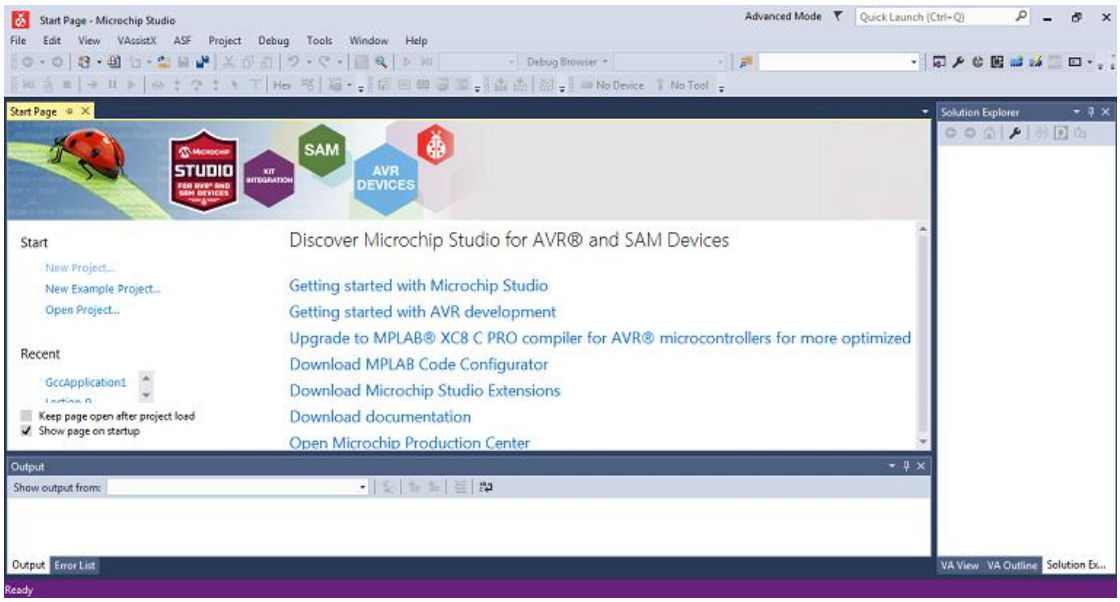


Рис. 1.

Для створення нового проекту потрібно в меню File, підменю Project, вибрати опцію меню New, як показано на рис. 2.

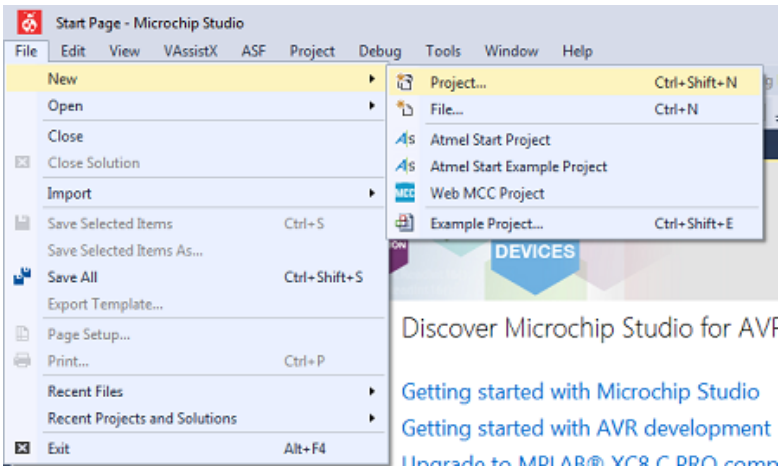


Рис. 2.

Відкриється діалогове вікно, що представлено на рис. 3.

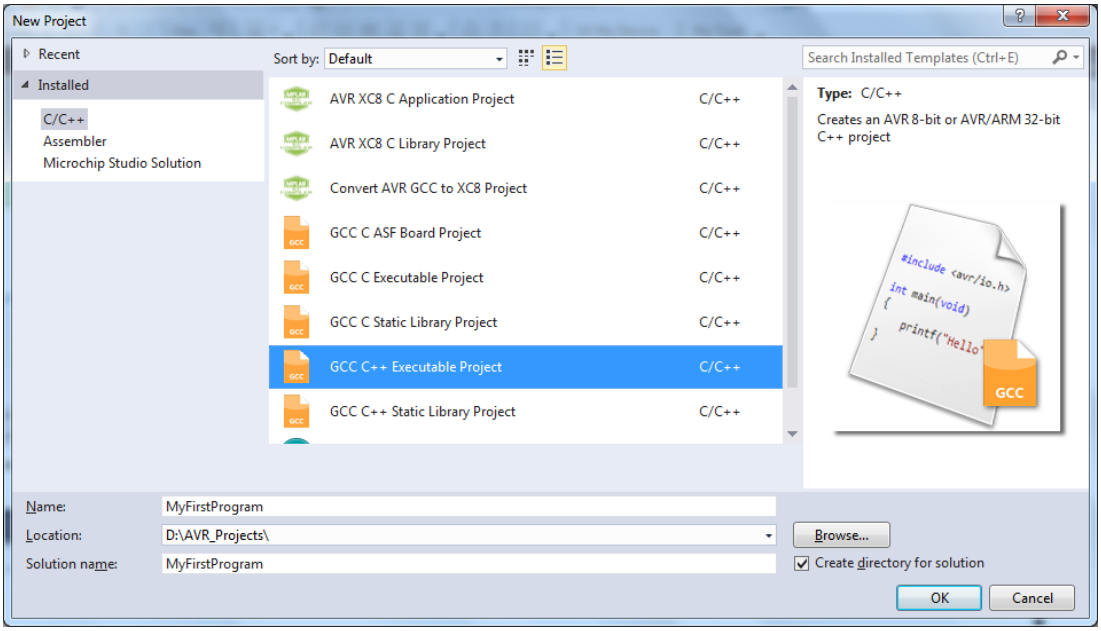


Рис. 3.

В цьому діалоговому вікні потрібно вибрати мову програмування C/C++ (у лівій частині вікна), компілятор та тип проекту – GCC C++ Executable Project, та вказати назву проекту (у наведеному діалоговому вікні, в нижній його частині, це MyFirstProject). Також слід звернути увагу на шлях до каталогу (поле Location), в якому будуть збережені файли проекту та створені нові файли в процесі компіляції. Саме в цьому каталозі буде створено файл прошивки для мікроконтролера, що матиме розширення .hex, який далі буде використано при безпосередньому програмуванні мікроконтролера. Коли всі налаштування зроблені можна натискати на кнопку ОК в нижній правій частині діалогового вікна. Відкриється наступне діалогове вікно, показане на рис. 4.

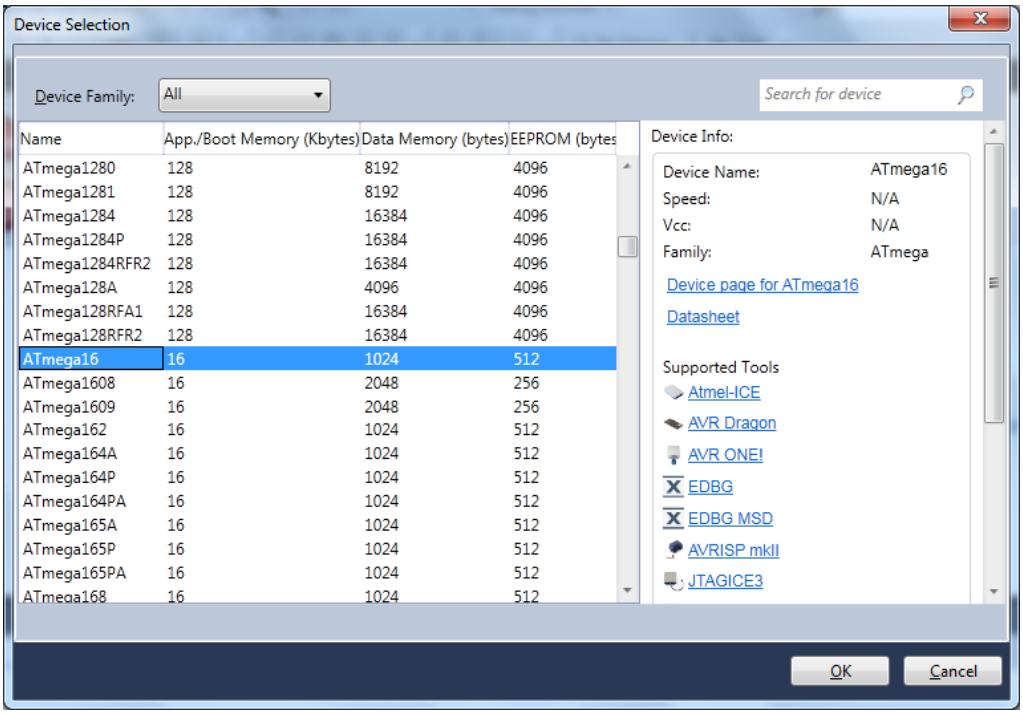


Рис. 4.

В цьому вікні у списку доступних мікроконтролерів потрібно вибрати Atmega16, адже саме цей мікроконтролер буде використовуватися далі, і натиснути на кнопку ОК. Середовищем розробки буде створено нові файли і відкриється вікно текстовий редактору, в якому буде створюватися код програми (рис. 5).

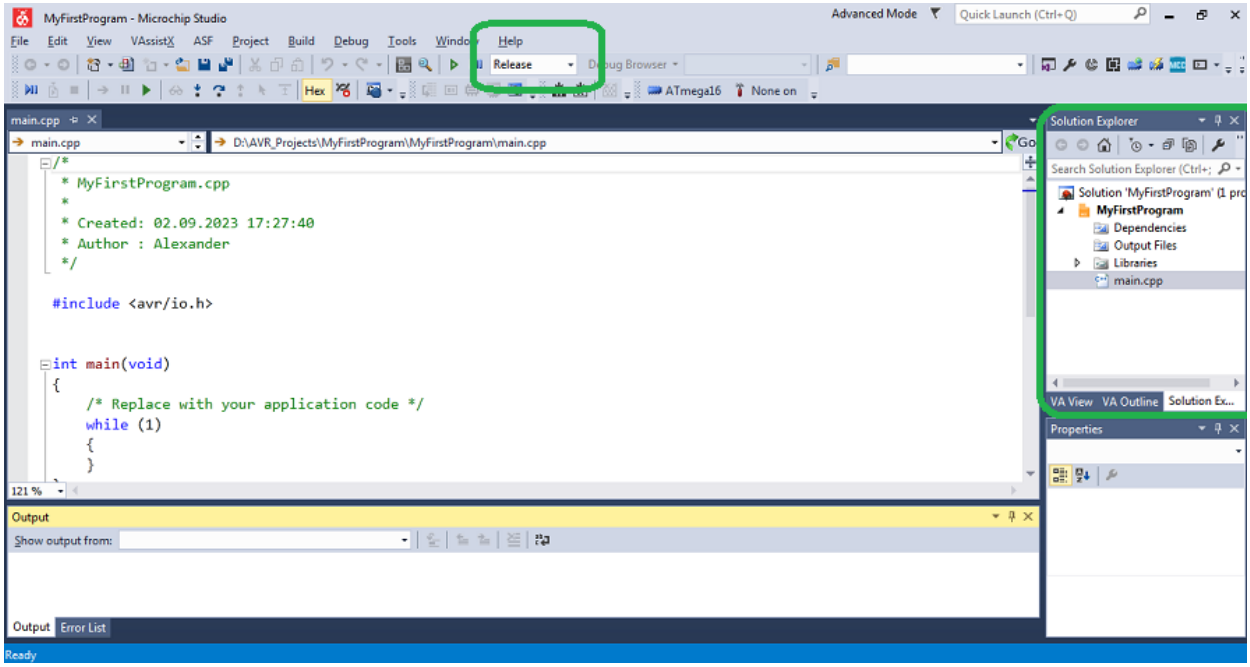


Рис.5.

В цьому вікні потрібно вибрати конфігурацію проекту – Debug або Release. В процесі розробки зазвичай користуються режимом Debug, а фінальна версія прошивки для мікроконтролера компілюється з конфігурацією Release. На рис. 5

показано де саме вибирається ця конфігурація. Також у правій частині вікна розташоване вікно Solution Explorer, в якому відображаються файли проекту. В даному прикладі проект містить лише один файл main.cpp. Для компіляції проекту потрібно в меню Build вибрати команду Build Solution, як показано на рис. 6.

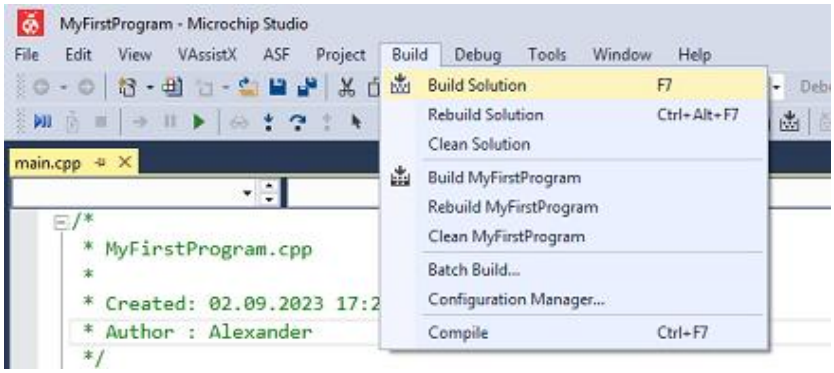


Рис. 6.

Після компіляції проекту у вікні Output буде відображено повідомлення про результати компіляції. Якщо в програмі немає синтаксичних помилок, то повідомлення буде мати вид, як показано на рис. 7.

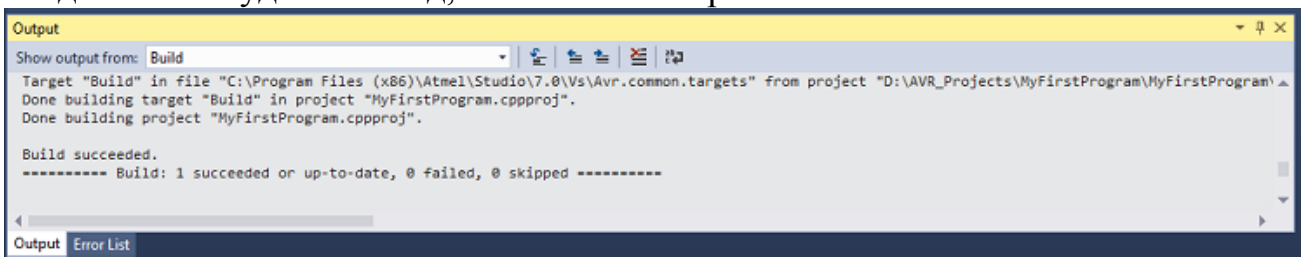


Рис. 7.

Якщо в програмі є помилки, то в результаті компіляції буде відображено повідомлення, як показано на рис. 8.

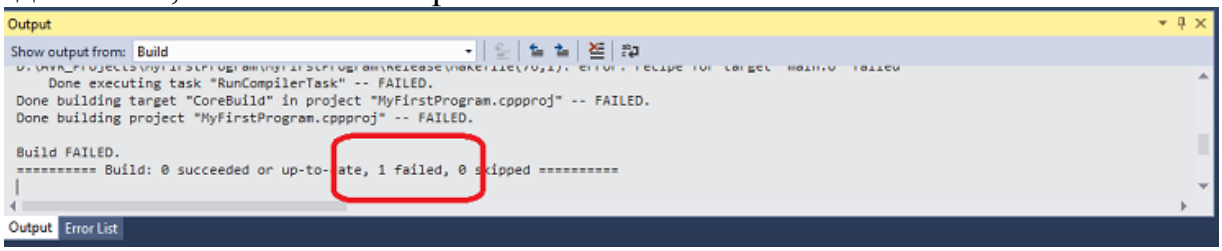


Рис. 8.

Перемкнувшись на вкладку Error List можна побачити інформацію про помилку та виправити її шляхом внесення змін до програмного коду (Рис. 9).

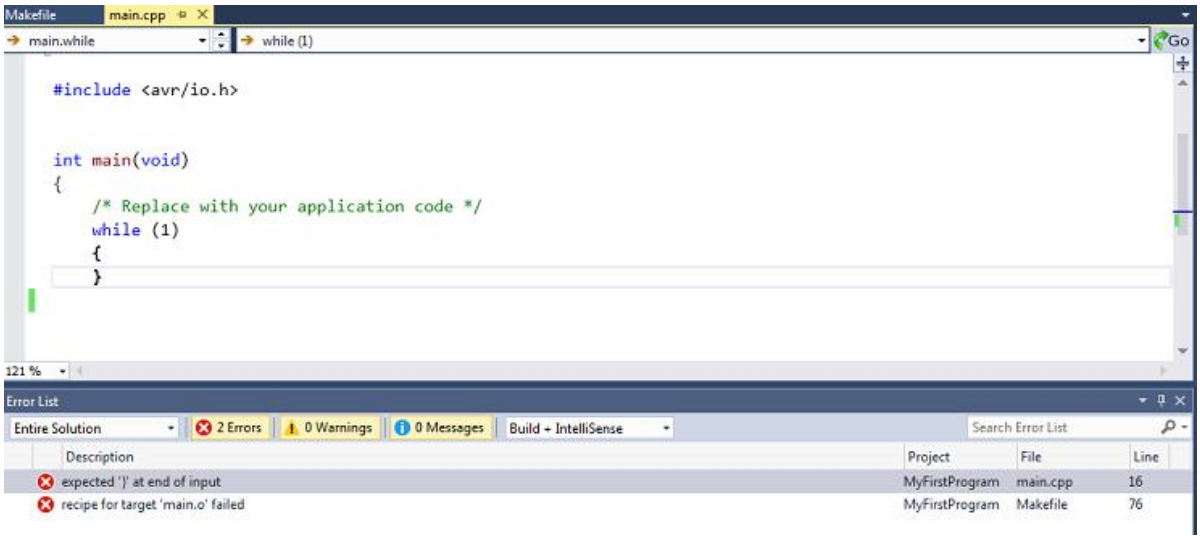


Рис. 9.

Двічі натиснувши на перше повідомлення в списку помилок курсор буде переміщено в місце програмного коду, що передує місцю помилки. Як видно з рисунку, в даному прикладі відсутня остання закриваюча фігурна дужка.

Після вдалої (тобто без помилок) компіляції проекту у каталозі, що було вказано при створенні проекту, з'являться нові файли, один з яких матиме розширення .hex, як показано на рис. 10. Саме цей файл буде потрібний в подальшому.

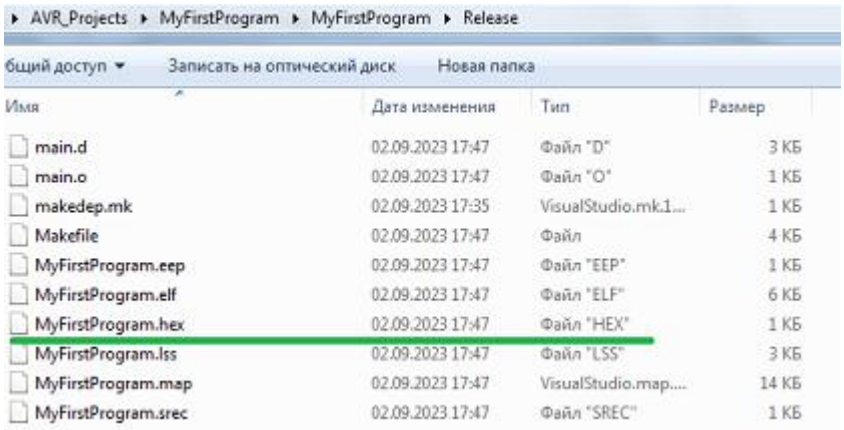


Рис. 10.

Мова програмування C++ на сьогоднішній день є однією з найпопулярніших і найпоширеніших, яка дозволяє створювати додатки для будь-якого спектру завдань: розробка прикладних додатків, мобільна розробка, системне програмування, програмування вбудованих мікропроцесорних систем. Відмінною особливістю програм на C++ є висока швидкість роботи (в порівнянні з Java, Python), тому ця мова програмування особливо часто використовуються в тих випадках, де необхідно забезпечити високу продуктивність і швидкодію.

Далі наведено приклад програми для мікроконтролера, що написана на мові C++ з використанням класів.

Приклад. В наступній програмі виконується розробка класу для роботи з портами вводу/виводу мікроконтролера ATMega16 та продемонстровано використання розробленого класу для створення ефекту «біжучий вогник» на

світлодіодах, що підключені до порту мікроконтролера. Фактично розроблений клас представляє собою драйвер портів мікроконтролера та надає високорівневий інтерфейс для керування портами вводу/виводу.

Спочатку за визначенням поняття "драйвер" – це функціональний компонент операційної системи, відповідальний за відносини з певною підмножиною апаратури комп'ютера.

У сучасному контексті це поняття трохи інше. Драйвер – це типовий інстанс-об'єкт (екземпляр) класу. Інтерфейс драйверів намагаються мінімізувати в термінах моделі read/write, але це марно. Наприклад, драйвер мережної карти має метод «прочитати MAC-адресу карти» (яку, звичайно, можна реалізувати через properties), а драйвер USB – цілий набір USB-специфічних операцій.

Цикл життя драйвера, в цілому, може бути описаний так:

– Ініціалізація: драйвер отримує ресурси (але не доступ до своєї апаратури).

– Пошук апаратури: драйвер одержує від ядра або знаходить сам свої апаратні ресурси.

– Активація – драйвер починає роботу.

– Поява/зникнення пристроїв, якщо це доречно.

– Вмикання режиму енергозбереження/вихід з режиму енергозбереження, якщо це доречно. У контролерах периферійних вузлів, що часто не використовується, вони вимикаються для економії.

– Деактивація драйвера – обслуговування запитів припиняється.

– Вивантаження драйвера – звільняються всі ресурси, драйвер не існує.

Зрозуміло, що в контексті вбудованих мікропроцесорних систем поняття драйвера дещо інше. В програмах, що будуть розглянуті в прикладах, під драйвером мається на увазі програмна оболонка (клас), що містить апаратно-залежну логіку, та надає відкритий інтерфейс доступу іншим об’єктам для взаємодії з апаратними ресурсами мікроконтролера. Таким чином, при зміні мікроконтролера не потрібно буде змінювати логіку поведінки програми, а потрібно буде лише змінити драйвер периферійного вузла, що у загальному випадку буде унікальним для кожного мікроконтролера. Хоча для мікроконтролерів AVR часто налаштування регістрів станів та керування периферійними вузлами схожі, і в окремих випадках клас-драйвер, розроблений для одного мікроконтролера може бути використаний для іншого мікроконтролера.

Програма починається з директив. Директиви – це спеціальні команди, які починаються з символу # і НЕ закінчуються крапкою з комою.

Директиву #define можна використовувати для створення макросів. Макрос – це правило, яке визначає конвертацію ідентифікатора у зазначені дані.

```
#define F_CPU 8000000
#define INPUT 0
#define OUTPUT 1
```

При використанні цих директив тепер в програмі є можливість використовувати замість значення 0 слово INPUT, а замість значення 1 слово OUTPUT. Такий підхід робить програмний код біль зрозумілим, адже, наприклад,

при налаштуванні порта вводу/виводу мікроконтролера на вихід в регістр DDRx (x – назва порта; для мікроконтролера ATMega16 це може бути A, B, C або D, тобто DDRA, DDRB, DDRC, DDRD) у відповідний розряд має бути записана 1. Якщо для налаштування порта вводу/виводу, а саме окремого виводу, на вхід або на вихід, використовується окрема функція, то значення 0 або 1, що визначає напрям роботи порта, має бути передане в якості аргументу функції. При використанні директив замість значення 0 або 1 в якості аргумента функції можна передавати INPUT або OUTPUT. В такому разі розробник відразу розуміє, як саме налаштований вивід мікроконтролера – на вхід або на вихід, що робить програмний код біль зрозумілим.

Також в програмі використовуються функції затримки. Для того. Щоб правильно розрахувати кількість тактів мікроконтролера для формування потрібної затримки. Компілятору потрібно знати на якій частоті працює мікроконтролер. Саме тому виконується оголошення макросу #define F_CPU 8000000. У глобальній змінній F_CPU, яка використовується в бібліотеці функцій формування часових затримок, міститься значення частоти роботи мікроконтролера (в Гц), яка в даному випадку складає 8 МГц.

Директива #include <filename> повідомляє препроцесору шукати файл у системних шляхах (у місцях зберігання системних бібліотек мови C++). Найчастіше буде використовуватися ця директивапри підключенні файлів заголовків з бібліотеки C++.

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

Клас призначений для опису певного типу об'єктів, в даному прикладі порта вводу/виводу мікроконтролера. Тобто, по суті, клас є планом об'єкта. А об'єкт представляє безпосереднє здійснення класу, його реалізацію. Для визначення класу використовується ключове слово class, після якого йде ім'я класу. Після назви класу у фігурних дужках розташовуються компоненти класу. Причому після фігурної дужки, що закриває клас, обов’язково йде крапка з комою. В даному прикладі клас називається Port. Як правило, назви класів починаються з великої літери. Клас може визначати змінні та константи (поля класу) для зберігання стану об'єкта та функції-члени класу (методи класу) для визначення поведінки об'єкта. Отже, драйвер портів вводу/виводу мікроконтролера реалізується як клас, що має методи (функції-члени класу) для керування режимами роботи портів:

```
class Port
{
private:
char m_name;
bool m_isOut;

public:
Port();
Port(char name, bool isOut);

void write(char value);
```

```
char read();

void set(char pos);
void reset(char pos);
};
```

Клас Port має два поля (змінні-члени класу) m_name та m_isOut, які призначені для зберігання імені порта (поле m_name містить один з символів: A, B, C або D для мікроконтролера ATMega16, що задається при створенні екземпляру класу) та режиму роботи порта – на вхід (якщо m_isOut має значення false) або вихід (якщо m_isOut має значення true). Змінні класи ще називають полями класу. Також клас визначає функцію print, яка виводить значення змінних класу на консоль. Також варто звернути увагу на модифікатор доступу public:, який вказує, що змінні, що йдуть після нього, і функції будуть доступні ззовні, із зовнішнього коду.

Клас може визначати різний стан, різні функції. Однак не завжди бажано, щоб до деяких компонентів класу був прямий доступ ззовні. Для розмежування доступу до різних компонент класу застосовуються специфікатори доступу

Специфікатор public робить члени класу – поля та функції відкритими, доступними з будь-якої частини програми. Однак, за допомогою іншого специфікатора private ми можемо приховати реалізацію членів класу, тобто зробити їх закритими, інкапсулювати всередині класу. В даному класі всі поля класу (тобто m_name та m_isOut) є закритими і не доступні для безпосередньої зміни ззовні – лише функції-члени цього класу мають доступ до цих полів. Натомість, всі конструктори та функції цього класу є відкритими, і саме шляхом виклику цих функцій-членів класу відбувається взаємодія з класом та зміна режимів роботи порта мікроконтролера та значень, що в нього виводяться.

Конструктори класу представляють спеціальну функцію, яка має те саме ім'я, що і клас, але яка не повертає жодного значення і яка дозволяє ініціалізувати об'єкт класу під час створення, і таким чином гарантувати, що поля класу будуть мати певні значення. При кожному створенні нового об'єкта класу викликається конструктор класу.

Тут під час створення об'єкта класу Port викликається конструктор за замовченням. Якщо в класі не визначено конструктор явно, то компілятор автоматично компілює конструктор за замовчуванням. Подібний конструктор не набирає жодних параметрів. В даному конструкторі за замовчуванням поля класу ініціалізуються значеннями – ім’я порта A (m_name = 'A';), і всі виводи порта налаштовані на вхід (m_isOut = 0; DDRA = 0).

```
Port::Port()
{
m_name = 'A';
m_isOut = 0;

DDRA = 0;
}
```

Якщо при створенні об’єкта порта потрібно вказати його назву та напрям роботи, використовується перевантажений конструктор. По суті, перевантажений

конструктор представляє собою функцію, яка може приймати параметри і яка повинна називатися так само, як і ім'я класу. В даному випадку конструктор приймає два параметри (name та isOut) і передає їх значення полям m_name та m_isOut. При цьому в залежності від значення другого параметра конструктора (змінна isOut яка копіюється в поле m_isOut) виконується налаштування відповідного порту на вхід (якщо isOut має значення false) або на вихід (якщо isOut має значення true). Перевантажені конструктори полегшують створення кількох об'єктів, які повинні мати різні значення.

```
Port::Port(char name, bool isOut)
{
m_name = name;
m_isOut = isOut;

switch(m_name)
{
	case 'A':
		DDRA = (isOut == 0)? 0 : 0xFF;
	break;
	case 'B':
		DDRB = (isOut == 0)? 0: 0xFF;
	break;
	case 'C':
		DDRC = (isOut == 0)? 0: 0xFF;
	break;
	case 'D':
		DDRD = (isOut == 0)? 0: 0xFF;
	break;
}
}
```

Конструкція switch-case дозволяє порівняти деякий вираз із набором значень.

```
switch (вираз)
{
     case значення_1: інструкції_1;
     case значення_2: інструкції_2;
     ..................
     case значення_N: інструкції_N;

     default: інструкції;
}
```

Після ключового слова switch у дужках йде порівнюваний вираз. Значення цього виразу послідовно порівнюється зі значеннями після оператора case. І якщо збіг буде знайдено, то виконуватиметься певний блок case. Варто зазначити, що вираз, що порівнюється в switch повинен представляти один з цілочисленних або символьних типів. Наприкінці конструкції switch може стояти блок default. Він необов'язковий і виконується в тому випадку, якщо значення після switch не відповідає жодному оператору case. Щоб уникнути виконання наступних блоків case/default, наприкінці кожного блоку ставиться оператор break. Після виконання

оператора break відбудеться вихід із конструкції switch...case, та інші оператори case будуть проігноровані.

Тернарний оператор певною мірою схожий на конструкцію if-else. Він приймає три операнди у такому вигляді:

```
операнд_1 ? операнд_2 : операнд_3
```

Перший операнд представляє умову. Якщо ця умова вірна (дорівнює true), тоді вибирається/виконується другий операнд, який міститься після символу ?. Якщо умова не вірна, тоді вибирається/виконується третій операнд, який міститься після двокрапки.

В мові C++ можна розділяти оголошення й реалізацію конструкторів та функцій, зокрема функцій, що створюються у класах. Для цього використовується такий вираз:

```
ім'я_класу::ім'я_функції(параметри) { тіло_функції}.
```

В даному прикладі всі конструктори та функції-члени класу (методи класу), що оголошені всередені класу, реалізовані (визначені) за його межами. Завданням функції write є запис значення в порт. При цьому в який саме порт буде записане значення value, що передається в якості параметра функції, визначається назвою порта, яка була вказана при створенні об’єкту класу та зберігається в полі m_name класу. Параметр функції має тип даних char. Як відомо, розрядність цього типу даних складає 8 біт, і мікроконтролер ATMega16, що використовується в даному прикладі, також 8-розрядний і має 8-розрядні порти. Саме тому розрядність використаних даних в даному випадку складає 8 біт.

```
void Port::write(char value)
{
switch(m_name)
{
       case 'A':
        PORTA = value;
       break;
       case 'B':
        PORTB = value;
       break;
       case 'C':
        PORTC = value;
       break;
       case 'D':
        PORTD = value;
       break;
}
}
```

Не менш важливим є функція, що повертає значення, в якому міститься інформація про рівні сигналів на виводах порта. Ця функція-член класу надає можливість зчитування даних з порта та повертає результат у вигляді 8-розрядного значення (саме тому тип даних що повертається цією функцією

вказано як char). Ця функція може бути використана для зчитування даних з дискретних вимикачів (кнопок), цифрових датчиків та ін.

```
char Port::read()
{
char value = 0;

if(m_name == 'A')
{
	value = PINA;
}
else if(m_name == 'B')
{
	value = PINB;
}
else if(m_name == 'C')
{
	value = PINC;
}
else if(m_name == 'D')
{
	value = PIND;
}

return value;
}
```

Наступна функція-член класу дозволяє змінити стан окремого виводу порта (встановити високий рівень сигналу (лог. 1)), номер якого передається в якості аргументу, при цьому не впливаючи на інші виводи. Це значення визначає номер виводу (розряд біту в регістрі керування портом). Оскільки порти мікроконтролера 8-розрядні, то номер виводу може приймати значення від 0 до 7. Для зміни значення лише одного розряду, а саме встановлення його значення лог.1, використовуються бітові операції зсуву та логічного АБО.

```
void Port::set(char pos)
{
switch(m_name)
{
	case 'A':
		PORTA |= (1<<pos);
	break;
	case 'B':
		PORTB |= (1<<pos);
	break;
	case 'C':
		PORTC |= (1<<pos);
	break;
	case 'D':
		PORTD |= (1<<pos);
	break;
}
}
```

Ця функція-член класу є ‘дзеркальною’ від попередньої і дозволяє встановити на окремму виводі порта рівень лог. 0. Номер виводу передається в

якості аргументу функції. Номер виводу також може приймати значення від 0 до 7.

```
void Port::reset(char pos)
{
	switch(m_name)
	{
		case 'A':
		PORTA &= ~(1<<pos);
		break;
		case 'B':
		PORTB &= ~(1<<pos);
		break;
		case 'C':
		PORTC &= ~(1<<pos);
		break;
		case 'D':
		PORTD &= ~(1<<pos);
		break;
	}
}
```

Функція main повинна бути в кожній програмі і є точкою входу, тобто саме з виконання цієї функції починається робота програми. В цій функції створюється екземпляр класу Port з назвою ledPort, що є об'єктом класу. При створенні цього об'єкту було вказано два параметри – назва порта 'C' та режим його роботи OUTPUT (на вихід). Також створено додаткову змінну з назвою «i». В циклі while виконується передавання значення змінної 'i' в функцію-член класу шляхом виклику метода set (установка високого рівня на виводі порта) класу Port через об'єкт ledPort, формування часової затримки 250 мс, встановлення рівня лог. 0 на тому самому виводі шляхом виклику метода reset та формування ще однієї затримки 250 мс. Після цього відбувається збільшення змінної 'i' на одиницю за умови, що поточне значення менше 7 (інакше змінна онулюється) і цикл повторюється. Таким чином, в програмі реалізовано створення ефекту «біжучий вогник» на світлодіодах.

```
int main(void)
{
    Port ledPort('C', OUTPUT);
    char i = 0;

    while (1)
    {
      // Running light//
      ledPort.set(i);
      _delay_ms(250);
      ledPort.reset(i);
      _delay_ms(250);

      if(i<7) i++;
      else i=0;
    }
}
```

Як було зазначено вище, розроблений клас має два члени (поля) для збереження назви порта та режиму його роботи – на вхід або на вихід. Таким чином, в кожен момент часу використання драйвера за значенням поля, що містить ім’я порта, буде зрозуміло, в який з регістрів керування потрібно записувати дані, щоб змінити режим роботи порта та рівні сигналів на його виводах.

На рис. 11 представлено принципову схему в Proteus, за допомогою якої можна промоделювати роботу системи та перевірити роботу програми.

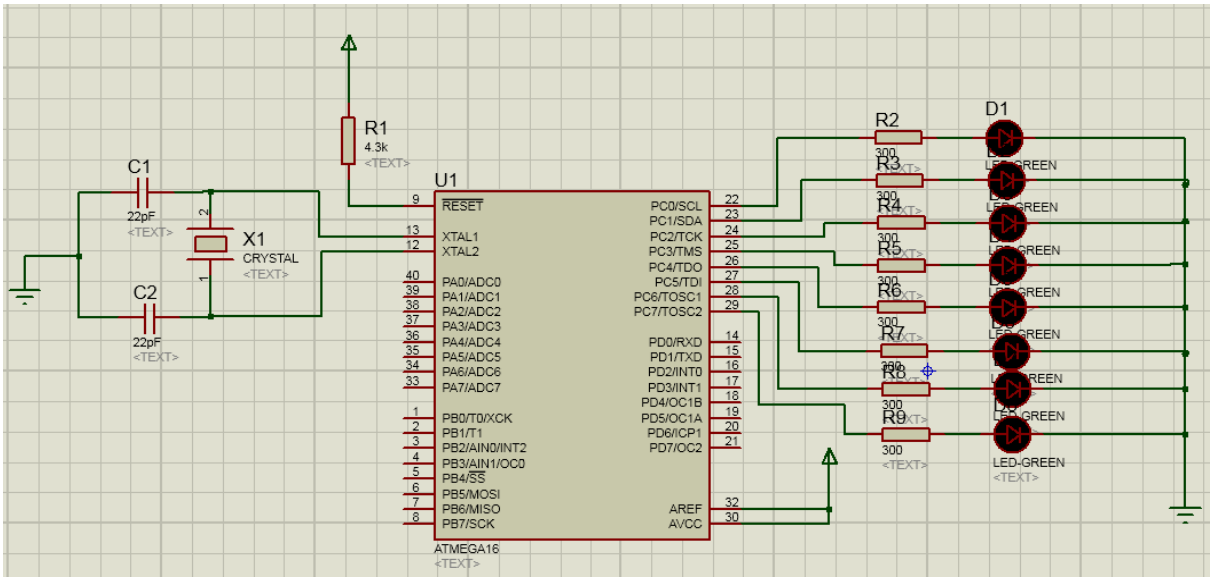


Рис. 11.

Завдання для самостійного виконання:

1. Встановити середовище розробки Microchip Studio 7, яке можна завантажити за посиланням:

https://atmel-studio.software.informer.com/download/?lang=ru

2. Встановити середовище моделювання електронних схем Proteus.

3. Прочитати «Розділ 10. Об’єкти і класи » (стор. 484-535) в підручнику «Стивен Прата. Мова програмування С++ (6 видання)» (підручник на гугл-диску):

https://drive.google.com/drive/u/0/folders/1MiXpe2uS_x0g-2UJGB89kGaOlhcDjzGi

4. Нижче наведено опис (декларація) класу Person:

```
class Person {
private:
constexpr LIMIT = 25;
string mLastName; // Person's last name
char mFirstName[LIMIT]; // Person's first name
public:
Person() {
mLastName = "";
mFirstName[0] = '\0';
```

```
} // #1
Person(const string & ln, const char * fn = "Heyyou"); // #2
// the following methods display mLastName and mFirstName
void Show() const; // firstname lastname format
void FormalShow() const; // lastname, firstname format
};
```

Завершіть реалізацію методів класа та наведіть приклад використання класу в програмі (Visual Studio).

5. Нижче наведено опис (декларація) класу Move:

```
class Move
{
private:
    double x;
    double y;
public:
    Move(double a = 0, double b = 0);   // sets x, y to a, b
    showmove() const;                   // shows current x, y values
    Move add(const Move & m) const;
// this function adds x of m to x of invoking object to get new x,
// adds y of m to y of invoking object to get new y, creates a new
// move object initialized to new x, y values and returns it
    reset(double a = 0, double b = 0);  // resets x,y to a, b
};
```

Створіть функції-члени класу та приклад програми з використанням цього класу (Visual Studio).

6. В середовищі програмування Microchip Studio потрібно розробити клас для роботи з семисегментним індикатором, що підключається до мікроконтролера ATMega16 (схема підключення семисегментного індикатора – із загальним катодом). Реалізвати конструктор класу, методи для керування індикатором. Написати програму, в якій використовується розроблений клас, для виводу на семисегментний індикатор чисел від 0 до 9 (лічильник від 0 до 9) із затримкою 1 секунда перед зміною значення. Інформація про семисегментний індикатор підручнику «СХЕМОТЕХНІКА ЕЛЕКТРОННИХ ПРИСТРОЇВ ТА СИСТЕМ. ТОМ 6. Апаратно-програмні засоби відображенні інформації», стор. 57-75.

7. Розробити в середовищі Proteus принципову схему пристрою та перевірити роботу програми.

8. Оформити звіт, в якому навести код програми та розроблену схему. Завантажити звіт на гугл-диск.

Практична робота 2.

Тема: Використання композиції та асоціації в об’єктно-орієнтованому програмуванні, розробка спрощеного класу (драйвера) порта вводу/виводу мікроконтролера. Приклад програми – клас, в якому реалізовано керування портом вводу/виводу мікроконтролера для створення різноманітних світлових ефектів (компілятор Atmel Studio 7).

Композиція – це важливий інструмент у програмуванні, який дає змогу створювати складніші системи з простих компонентів. Тобто, замість того щоб створювати новий об’єкт з нуля, можна використовувати вже наявні об’єкти та комбінувати їх у складніші структури (класи).

На рис. 1 представлено принципову схему моделі мікропроцесорної системи, за допомогою якої виконується перевірка роботи програм (файл Proteus_model_Lection_2.pdsprj на гугл-диску додається). В схемі використано мікроконтролер ATMega16, до порта С якого підключено 8 світлодіодів. Станом саме цих світлодіодів відбувається керування (вмикання та вимикання).

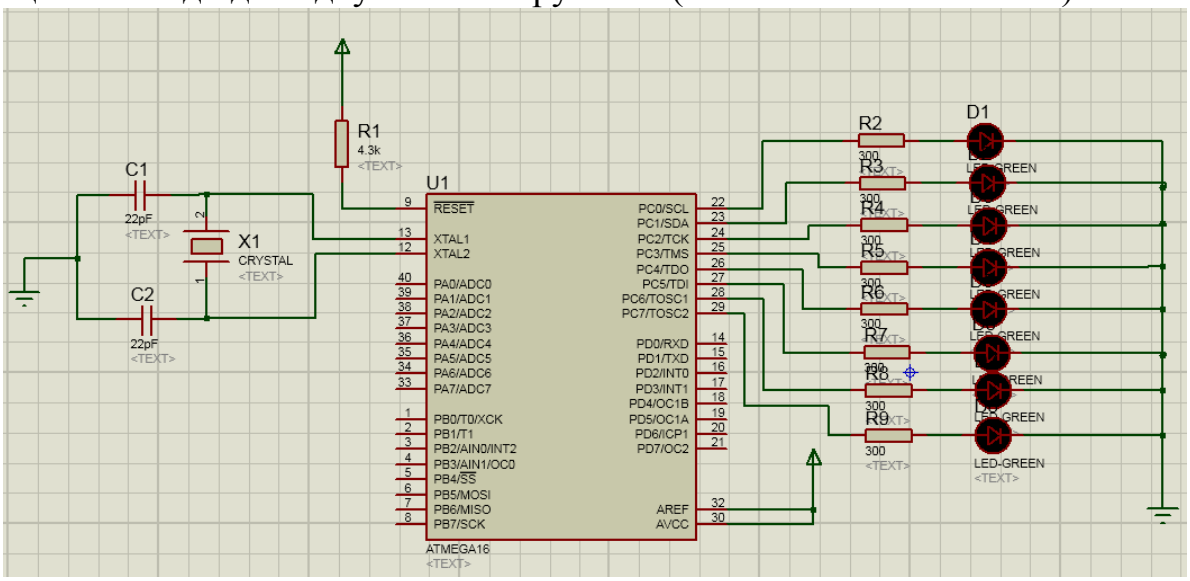
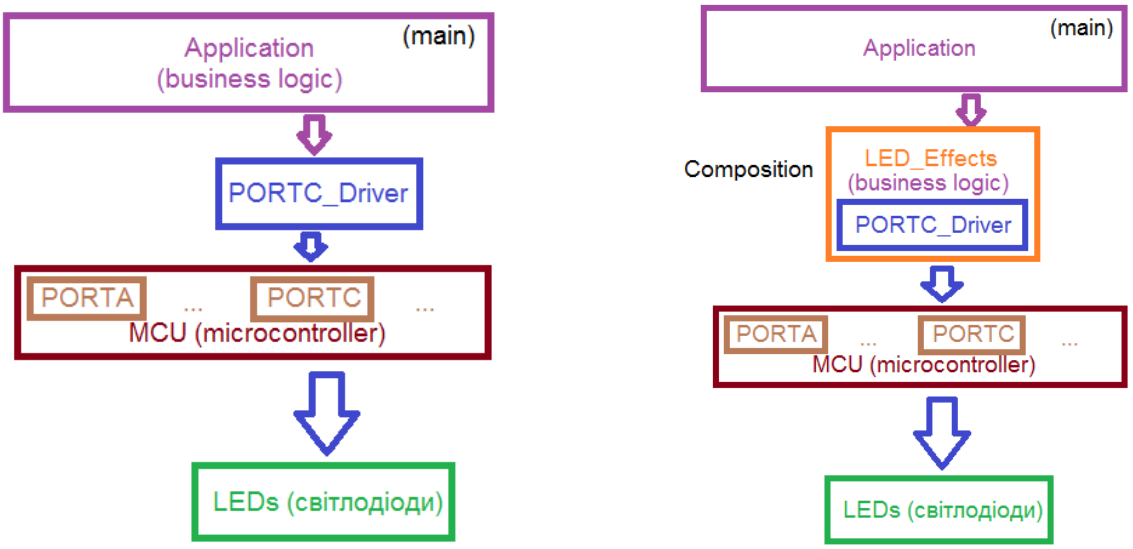


Рис. 1.

Приклад 1 (використання композиції). В цьому прикладі продемонстровано принцип використання композиці при створенні класів. Реалізовано 2 класи – клас PORTC_Driver для роботи з портом С мікроконтролера (фактично, клас являє собою спрощений варіант драйверу порта вводу/виводу мікроконтролера), та клас LED_Effects, в якому реалізовано формування різноманітних світлових ефектів. Клас для роботи з портом вводу/виводу PORTC_Driver є складовою частиною класу LED_Effects.

На рис. 2, б, представлено структуру програми, що розглядається в цьому прикладі.



а б

Рис. 2.

В порівнянні зі структурою програмного забезпечення, що представлена на рис. 2, а, в цьому прикладі (рис. 2, б) логіка керування станом порта вигесена в окремий клас LED_Effects, і в функції main() сворюється екземпляр цього класу. Шляхом виклику методів класу відбуваться керування портом вводу/виводу мікроконтролера. Композиція – це процес створення складних об’єктів шляхом комбінування та об’єднання простіших об’єктів. Композиція використовується для того, щоб розбити складну задачу на простіші підзадачі та вирішувати їх окремо, а потім комбінувати для отримання підсумкового результату. Це дає змогу підвищити модульність коду, поліпшити його читабельність і перевикористання. Крім того, композиція дає змогу створювати більш гнучкі та розширювані системи, оскільки окремі компоненти можуть бути легко замінені або змінені без впливу на решту коду. Це також допомагає уникнути дублювання коду, завдяки використанню одних і тих самих компонентів у різних місцях.

```
#define F_CPU 8000000

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

const int DELAY = 150;

class PORTC_Driver
{
public:
PORTC_Driver();
PORTC_Driver(bool isOutput);

void write(char data);
void setBit(char pos);
void resetBit(char pos);
};

PORTC_Driver::PORTC_Driver()
{
```

```
DDRC = 0b11111111; // 0xFF, 255
}

PORTC_Driver::PORTC_Driver(bool isOutput)
{
if(isOutput == false)
{
	DDRC = 0b00000000;
}
else
{
	DDRC = 0b11111111;
}
}

void PORTC_Driver::write(char data)
{
PORTC = data;
}

void PORTC_Driver::setBit(char pos)
{
if(pos <= 7)
{
	PORTC |= (1 << pos);
}
}

void PORTC_Driver::resetBit(char pos)
{
if(pos <= 7)
{
	PORTC &= ~(1 << pos);
}
}

class LED_Effects
{
PORTC_Driver m_port;

public:
LED_Effects();

void blinking_lights();
void running_lights();
void bumping_lights();
};

LED_Effects::LED_Effects() : m_port(true)
{	// Nothing to do here
}

void LED_Effects::blinking_lights()
{
m_port.write(0b11111111);
_delay_ms(DELAY);
m_port.write(0b00000000);
_delay_ms(DELAY);
}
```

```
void LED_Effects::bumping_lights()
{
char data[6] = {0b10000001, 0b01000010, 0b00100100, 0b00011000, 0b00100100, 0b01000010};
int i;
for(i = 0; i<6; i++)
{
	m_port.write(data[i]);
	_delay_ms(DELAY);
}
m_port.write(0b00000000);
}

void LED_Effects::running_lights()
{
int i;
for(i = 0; i < 8; i++)
{
	m_port.setBit(i);
	_delay_ms(DELAY);
	m_port.resetBit(i);
}
}

int main(void)
{
    LED_Effects myLEDEffect;

    while (1)
    {
      myLEDEffect.blinking_lights();
      myLEDEffect.blinking_lights();
      myLEDEffect.blinking_lights();

      myLEDEffect.bumping_lights();

      myLEDEffect.running_lights();
    }
}
```

В класі myLEDEffect реалізовано 3 методи: blinking_lights(), bumping_lights() та running_lights(). При виклику методу blinking_lights() відбувається вмикання та вимикання всіх світлодіодів через певний проміжок часу, що визначається значенням глобальної змінної DELAY. Пи виклику функції bumping_lights() вібдувається формування сітолвого ефекту вогників, що зіштовхуться. Рис. 3 допомагає зрозуміти суть цього алгориму.

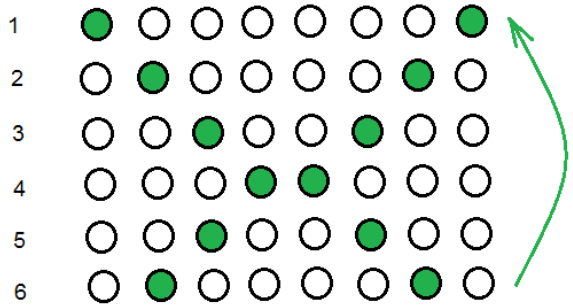


Рис. 3.

При виклику методу running_lights() формується ефект «біжучого вогника» (рис. 4).

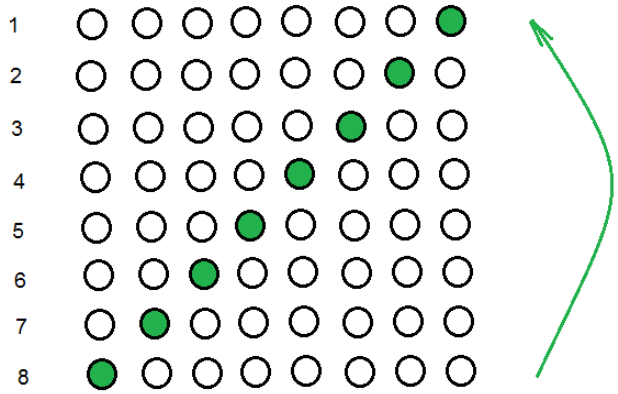


Рис. 4.

Приклад 2 (асоціація). Асоціація – це зв'язок між класами, а композиція – це вкладеність одного класу в інший, але при цьому клас обгортка не управляє терміном життя вкладеного об'єкта (який представлений посиланням на об'єкт, що шукається). На рис. 5 представлено структуру програми, що розглядається в даному прикладі і в якій використовується асоціація. На відміну від попереднього прикладу, в цій програмі клас LED_Effects містить не екземпляр класу PORTC_Driver (тобто, об’єкт для роботи з портов вводу/виводу), а вказівник на нього. Сам об’єкт для роботи з портом вводу/виводу (екземпляр класу PORTC_Driver) створюється за межами LED_Effects, і передається в конструктор класу LED_Effects за допомогою вказівника. Отже, екземпляр клас LED_Effects (об’єкт myLEDEffect) ніяк не керує життєвим циклом екземпляру класу PORTC_Driver (об’єкт myPortC). У цій концепції ООП усі об'єкти мають окремий життєвий цикл, і вони не мають власника.

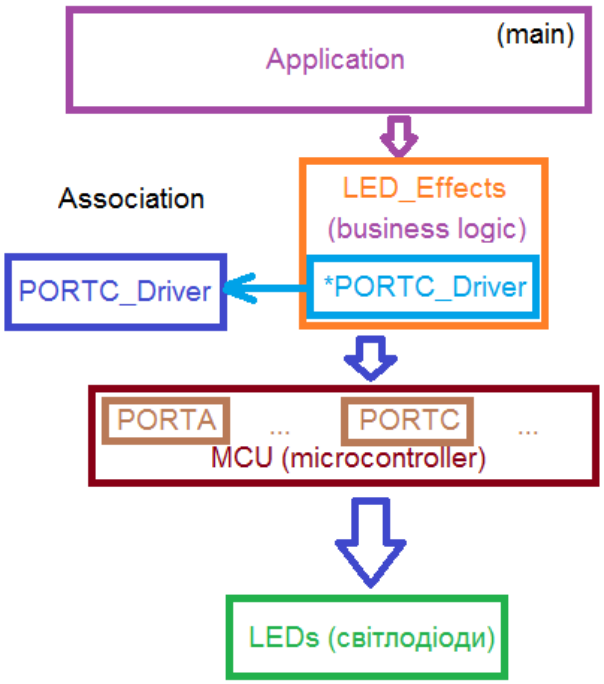


Рис. 5.

Далі наведено текст програми.

```
#define F_CPU 8000000

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

const int DELAY = 200;

class PORTC_Driver
{
public:
PORTC_Driver();
PORTC_Driver(bool isOutput);

void write(char data);
void setBit(char pos);
void resetBit(char pos);
};

PORTC_Driver::PORTC_Driver()
{
DDRC = 0b11111111; // 0xFF, 255
}

PORTC_Driver::PORTC_Driver(bool isOutput)
{
if(isOutput == false)
{
	DDRC = 0b00000000;
}
else
{
	DDRC = 0b11111111;
}
}

void PORTC_Driver::write(char data)
{
PORTC = data;
}

void PORTC_Driver::setBit(char pos)
{
if(pos <= 7)
{
	PORTC |= (1 << pos);
}
}

void PORTC_Driver::resetBit(char pos)
{
if(pos <= 7)
{
	PORTC &= ~(1 << pos);
}
}
```

```
class LED_Effects
{
PORTC_Driver *m_port;

public:
LED_Effects(PORTC_Driver *port);

void blinking_lights();
void running_lights();
void bumping_lights();
};

LED_Effects::LED_Effects(PORTC_Driver *port)
{
m_port = port;
}

void LED_Effects::blinking_lights()
{
m_port->write(0b11111111);
_delay_ms(DELAY);
m_port->write(0b00000000);
_delay_ms(DELAY);
}

void LED_Effects::bumping_lights()
{
char data[6] = {0b10000001, 0b01000010, 0b00100100, 0b00011000, 0b00100100, 0b01000010};
int i;
for(i = 0; i<6; i++)
{
	m_port->write(data[i]);
	_delay_ms(DELAY);
}
m_port->write(0b00000000);
}

void LED_Effects::running_lights()
{
int i;
for(i = 0; i < 8; i++)
{
	m_port->setBit(i);
	_delay_ms(DELAY);
	m_port->resetBit(i);
}
}

int main(void)
{
    PORTC_Driver myPortC;
    LED_Effects myLEDEffect(&myPortC);

    while (1)
    {
       myLEDEffect.blinking_lights();
       myLEDEffect.blinking_lights();
       myLEDEffect.blinking_lights();

       myLEDEffect.bumping_lights();
```

```
        myLEDEffect.running_lights();
    }
}
```

Отже, між двома класами/об'єктами є різні типи відносин. Найбільш базовим типом відносин є асоціація (association), це означає, що два класи якось пов'язані між собою. Різниця між композицією та асоціацією(агрегацією) полягає в тому, що у разі композиції ціле явно контролює час життя своєї складової частини (частина не існує без цілого), а у разі асоціації ціле хоч і містить свою складову частину, час їх життя не пов'язаний (наприклад, складова частина передається через параметри конструктора).

Агрегація, або її ще називають «слабка асоціація» – це такий тип відносин, коли клас має об'єкт, який він запозичив десь ще (наприклад в іншого класу). Композиція, чи «сильна асоціація» – це такий тип відносин, коли клас саме «володіє» об'єктом і відповідає за його час життя.

Цікавою особливістю різних відносин між класами і те, що логічність їх використання може залежати від погляду проектувальника, від того, з якого боку дивиться на завдання і які питання він ставить собі при аналізі. Саме тому одну і ту ж задачу можна вирішити десятком різних способів, при цьому в одному випадку ми отримаємо сильно пов'язаний дизайн з великою кількістю композиції (приклад 1), а в іншому випадку (приклад 2) – це завдання буде розбито на більш автономні будівельні блоки, що поєднуються між собою з за допомогою асоціації.

Завдання для самостійного виконання:

1. Прочитати «Розділ 11. Робота з класами» (стор. 535-590) в підручнику «Стивен Прата. Мова програмування С++ (6 видання)» (підручник на гугл-диску) та параграф «3.2. Паралельні порти вводу-виводу» (стор. 91-93) в підручнику «Рябенький В. М., Ушкаренко О. О., Буряк В. С. Схемотехніка електронних пристроїв та систем. Том 3. Мікропроцесорна техніка».
https://drive.google.com/drive/u/0/folders/1MiXpe2uS_x0g-2UJGB89kGaOlhcDjzGi

2. Розширити клас LED_Effects, наведений у прикладах, шляхом додавання ще 2-х методів, в кожному з яких реалізовано окремий світлови ефект (наприклад, «біжуча тінь», «гусениця» або інший, на васний розсуд). Промоделювати роботу програми у середовищі Proteus та впевнитися, що результати реалізації відповідають очікуванням.

3. Модифікувати схему шляхом підключення до порту D мікроконтролера ще 8 срикладі для порту C). Внести зміни в програмний код так, щоб клас LED_Effects міг керувати портом D мікроконтролера. Перевірити роботупрограми шляхом моделювання роботи пристрою системи в Proteus.

4. Оформити звіт, в якому навести код програм та схему. Завантажити звіт на гугл-диск.

Практична робота 3.

Тема: Використання архітектурного паттерну Модель-Представлення-Контролер (Model-View-Controller, MVC) при реалізації програм для мікроконтролерів. Приклад програми – лічильник імпульсів (драйвер портів мікроконтролера ATMega16, клас лічильника, клас для виводу значення лічильника в порт; використано архітектурний паттерн MVC, компілятор Microchip Studio 7).

Зважаючи на мету зменшення трудовитрат на розробку складного програмного забезпечення, припустимо, що необхідно використовувати готові уніфіковані рішення. Адже шаблонність дій полегшує комунікацію між розробниками, дозволяє посилатися на відомі конструкції, знижує кількість помилок. Паттерн (англ. design pattern) – архітектурна конструкція, що представляє собою вирішення проблеми проектування в рамках деякого контексту, що часто виникає.

Модель-представлення-контролер (MVC, англ. Model-view-controller) – архітектурний шаблон (паттерн проектування), який використовується під час проектування та розробки програмного забезпечення.

Модель (Model)

Під Моделлю зазвичай розуміється та частина програмного забезпечення, яка містить у собі функціональну бізнес-логіку. Модель має бути повністю незалежною від інших елементів програми. Модельний шар нічого не повинен знати про елементи дизайну, і яким чином він відображатиметься. Досягається результат, що дозволяє змінювати представлення даних не чіпаючи саму Модель.

Модель має такі ознаки:

– модель – це бізнес-логіка програми;

– модель має знання про себе саму і не знає про контролерів і уявлення;

– для деяких проектів модель це просто шар даних (клас, що містить лише поля та геттери/сеттери для них; база даних; файл);

– для інших проектів модель — менеджер бази даних, набір об'єктів або просто логіка програми.

Модель інкапсулює ядро даних і основний функціонал їхньої обробки та не залежить від процесу вводу чи виводу даних.

Представлення (View)

До обов'язків Представлення входить відображення даних, отриманих від Моделі. Однак Представлення не може безпосередньо впливати на модель. Можна говорити, що представлення має доступ лише зчитування даних.

Представлення має такі ознаки:

– у Представленні реалізується відображення даних, що виходять від моделі будь-яким способом;

– у деяких випадках Представлення може мати код, який реалізує бізнес-логіку.

Контролер (Controller)

Контролер одержує вхідні дані й перетворює їх на команди для моделі чи вигляду. У функції контролера входить відстеження визначених подій, що

виникають в результаті дій користувача. Контролер дозволяє структурувати код шляхом групування пов'язаних дій в окремий клас. Зареєстровані події транслюються в різні запити, що спрямовуються компонентам моделі або об'єктам, відповідальним за відображення даних. Відокремлення моделі від вигляду даних дозволяє незалежно використовувати різні компоненти для відображення інформації. Таким чином, якщо користувач через контролер внесе зміни до моделі даних, то інформація, подана одним або декількома візуальними компонентами, буде автоматично відкоригована відповідно до змін, що відбулися. На рис. 1 представлено структуру паттерну MVC.

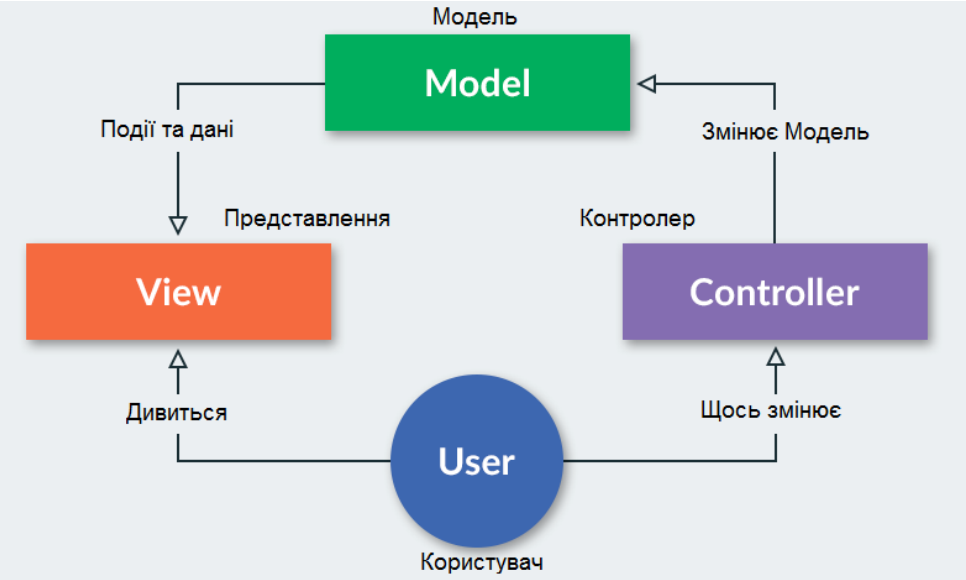


Рис. 1.

Приклад програми – реалізація двійкового лічильника з використанням паттерну MVC.

Як і раніше, програма починається з визначення значення глобальної змінної *F_CPU*, в якій зберігається значення частоти роботи мікроконтролера (8 МГц), за допомогою директиви #define. Ця змінна використовується в бібліоі функцій фрмування часових затримок. Власне, підключення самих бібліотек, що використовуються в програмі, вібдувається за допомогою директиви #include. Також, щоб зробити код програми більш зрозумілим, часто числовим значенням дають «псевдоними». В даному прикладі замість значення 0 в програмі можна писати INPUT, а замість значення 1 відповідно OUTPUT. Це також зроблено за допомогою директиви #define.

```
#define F_CPU 8000000

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

#define INPUT 0
#define OUTPUT 1
```

Наступний клас використовується для керування портом вводу/виводу мікроконтролера. Клас містить два закритих поля – назву порта (m_name) та режим роботи (m_isOut; якщо false, то порт налаштовано на вхід, якщо true, то на вихід). Значення цих полів передаються в конструктор класу при створенні екземпляру класу. При цьому в класі реалізовано конструктор за замовчуванням, і якзо при створенні об’єкта не було передано ніяких параметрів, то за замовчуванням ім’я порта матиме значення ‘A’, і він буде налаштований на вхід (m_isOut = 0;).

```
class Port
{
private:
char m_name;
bool m_isOut;

public:
Port();
Port(char name, bool isOut);

void write(char value);
char read();

void set(char pos);
void reset(char pos);

bool getPinState(char pinNum);
};
```

Реалізація конструктора за замовчуванням.

```
Port::Port()
{
m_name = 'A';
m_isOut = 0;

DDRA = 0;
}
```

Реалізація перевантаженого конструктора класу. Параметрами конструктора виступають назва порту та режим його роботи. Передані параметри копіюються у відповідні поля класу. В залежності від переданих в перевантажений конструктор класу значень виконується запис у відповідний регістр налаштування режиму роботи порта значення 0 якщо порт налаштований на вхід, або 0xFF, якщо порт налаштовано на вихід. Для вибору потрібного регістра порта в заленості від переданого в конструктор параметру (імені порта) використано конструкцію switch-case (альтернатива if-else). При цьому використано тернарний оператор при налаштуванні режиму роботи порта та вибору, яке саме значення потрібно записати в регістр керування.

```
Port::Port(char name, bool isOut)
{
m_name = name;
m_isOut = isOut;
```

```
switch(m_name)
{
	case 'A':
		DDRA = (isOut == 0)? 0 : 0xFF;
	break;
	case 'B':
		DDRB = (isOut == 0)? 0: 0xFF;
	break;
	case 'C':
		DDRC = (isOut == 0)? 0: 0xFF;
	break;
	case 'D':
		DDRD = (isOut == 0)? 0: 0xFF;
	break;
}
}
```

Наступна функція-член (метод) класу використовується для того, щоб записати деяке 8-розрадне значення в порт. Як було зазначено раніше, порт мікроконтролера є 8-рорядним, тому тип даних параметру цього метода має char. В залежності від назви порта (A, B, C або D), відбувається запис переданого в функцію-член класу параметра у відповідний регістр порта, що призвоить дозміни рівнів сигналів на виводах мікроконтролера: якщо розряд змінної має значення 0, на відповідному виводі мікроконтролера встановлюється 0В, а якщо 1, то 5В (або 3,3В в залежності від напруги живлення).

```
void Port::write(char value)
{
switch(m_name)
{
	case 'A':
		PORTA = value;
	break;

	case 'B':
		PORTB = value;
	break;

	case 'C':
		PORTC = value;
	break;

	case 'D':
		PORTD = value;
	break;
}
}
```

Цей метод класу використовується для отримання 8-розядного значення (саме тому тип поверненого цим методом класу значення є char) що відповідає рівням сигналів на виводах порта (зазвичай при цьому порт налаштований на вхід). В залежності від імені порта (змінна m_name, значення якої буде передано в конструктор при створенні екземпляру класу), відбувається зчитування даних з відповідного регістру.

```
char Port::read()
{
char value = 0;

if(m_name == 'A')
{
	value = PINA;
}
else if(m_name == 'B')
{
	value = PINB;
}
else if(m_name == 'C')
{
	value = PINC;
}
else if(m_name == 'D')
{
	value = PIND;
}

return value;
}
```

Дуже корисною є функція-член класу, яка дозволяє змінити рівень сигналу лише на окремому виводі мікроконтролера (встановити високий рівень), при цьому не впливаючи на стани інших виводів цього самого порта. Номер виводу порта, на якому потрібно встановити рівень лог.1, передається в якості параметра цього метода класу. Далі, в залежності від назви порта (поле m_name), відбувається виконання бітової операції АБО вмісту відповідного регістра порта та значення маски, що отримується шляхом зсуву 1 на кількість розрядів pos (номер виводу – параметр функції) ліворуч.

```
void Port::set(char pos)
{
switch(m_name)
{
	case 'A':
		PORTA |= (1<<pos);
	break;

	case 'B':
		PORTB |= (1<<pos);
	break;

	case 'C':
		PORTC |= (1<<pos);
	break;

	case 'D':
		PORTD |= (1<<pos);
	break;
}
}
```

Не менш корисною є функція-член класу, яка дозволяє встановити низький рівень сигналу на окремому виводі мікроконтролера, при цьому не впливаючи на стани інших виводів цього самого порта. Номер виводу порта, на якому потрібно встановити рівень лог.0, передається в якості параметра цього метода класу. Далі, в залежності від назви порта (поле m_name), відбувається виконання бітової операції І вмісту відповідного регістра порта та значення маски, що отримується шляхом зсуву 1 на кількість розрядів pos (номер виводу – параметр функції) ліворуч та інверсії отриманого значення за допомогою оператора ~.

```
void Port::reset(char pos)
{
	switch(m_name)
	{
		case 'A':
			PORTA &= ~(1<<pos);
		break;

		case 'B':
			PORTB &= ~(1<<pos);
		break;

		case 'C':
			PORTC &= ~(1<<pos);
		break;

		case 'D':
			PORTD &= ~(1<<pos);
		break;
	}
}
```

Ця функція-член класу повертає значення false або true в залежності від рівня сигналу (лог.0 або лог.1) на окремому вході порта. Номер входу визначається параметорм pinNum, переданим у функцію. Саме цю функцію можна використовувати при необхідності зчитати інформацію про стан дискретних кнопок.

```
bool Port::getPinState(char pinNum)
{
bool result = false;
switch(m_name)
{
	case 'A':
		result = (PINA & (1<<pinNum));
	break;

	case 'B':
		result = (PINB & (1<<pinNum));
	break;

	case 'C':
		result = (PINC & (1<<pinNum));
	break;

	case 'D':
		result = (PIND & (1<<pinNum));
```

```
        break;
}

return result;
}
```

Реалізація паттерну MVC починається з реалізації класу представлення. Цей клас містить вказівник на драйвер порта (тобто вказівник на екземпляр класу, що реалізує функціональність для роботи з портов ввиду/виводу та надає відповідні функції-члени класу для цього). Завдання класу представлення полягає лише у відображенні даних. Тобто клас має надавати інтерфейс (функції), що виконуєть відображення даних, в даному випадку на світлодіодах. Оскільки безпосереднє керування станами виводів порта займаєься драйвер, то клас представлення, по суті, в даному випадку є посередником, тобто додає абстракцію в реалізацію програми. З точки зору архітектури це добре, адже всі апаратно залежні речі знаходяться у класі драйвера.

```
class View
{
Port * m_pPort;
public:
View(Port * port);

void OutData(unsigned int value);
};
```

Реалізація конструктора класу: параметр, що передаєься, є вказівник на екземпляр класу дравера для роботи з портами воводу/виводу.

```
View::View(Port * port)
{
m_pPort = port;
}
```

Функція відображення даних. Параметр, що передається в цю функцію, далі передається у функцію зміни станів виводів порта, що викликається через вказівник на екземпляр класу –драйвера порта вводу/виводу.

```
void View::OutData(unsigned int value)
{
m_pPort->write((char)value);
}
```

Реалізація класу моделі. Цей клас має містити лише дані, а те, як вони вони будуть відображені ні входить до зони відповідальності цього класу. Такий підхід дозволяє відокремити самі дані від системи відображення інформації, і за необхідності, відображати дані на різних індикаторах. При цьому сам клас моделі не зміниться, що робить архітектуру прогарми гнучкою та більш надійною, а також відкритою до змін та розширення.

```
class Counter_Model
{
private:
unsigned int m_value;
```

```
unsigned int m_maxValue;
unsigned int m_minValue;
View *m_view;

public:

Counter_Model();
Counter_Model(unsigned int initValue, unsigned int minValue, unsigned int maxValue);

void setValue(unsigned int newValue);
void setMaxValue(unsigned int newMaxValue);
void setMinValue(unsigned int newMinValue);
void setView(View * view);

unsigned int getValue();
unsigned int getMinValue();
unsigned int getMaxValue();

void Increment();
void Decrement();
void Reset();
};
```

В конструкторі за замовчуванням значення всіх полів класу встановлюються в нуль.

```
Counter_Model::Counter_Model()
{
 m_value=0;
 m_minValue=0;
 m_maxValue=0;
}
```

В перевантаженому конструкторі значення, що передані в якості аргументів, аписуються у відповідні поля класу.

```
Counter_Model::Counter_Model(unsigned int initValue, unsigned int minValue, unsigned int
maxValue)
{
m_value=initValue;
m_minValue=minValue;
m_maxValue=maxValue;
}
```

У функції, що дозволяє встановити поточне значення лічильника, також відбувається перевірка переданого значення, щоб воно не виходило за припустимі межі. Це робить програму більш надійною та стійкою до неправильних зовнішніх даних.

```
void Counter_Model::setValue(unsigned int newValue)
{
if((newValue>=m_minValue) && (newValue<=m_maxValue))
{
 m_value = newValue;
 m_view->OutData(m_value);
}
}
```

Функція дозволяє встановити максимально можливе значення лічильника.

```
void Counter_Model::setMaxValue(unsigned int newMaxValue)
{
m_maxValue = newMaxValue;
}
```

Функція дозволяє встановити мінімально можливе значення лічильника.

```
void Counter_Model::setMinValue(unsigned int newMinValue)
{
m_minValue = newMinValue;
}
```

Функція повертає поточне значення лічильника.

```
unsigned int Counter_Model::getValue()
{
return m_value;
}
```

Функція повертає встановлене мінімально можливе значення лічильника.

```
unsigned int Counter_Model::getMinValue()
{
return m_minValue;
}
```

Функція повертає встановлене максимально можливе значення лічильника.

```
unsigned int Counter_Model::getMaxValue()
{
return m_maxValue;
}
```

Виклик наступної функції призводить до збільшення значення лічильника на 1 за умови, що воно менше максимально можливого.

```
void Counter_Model::Increment()
{
if(m_value < m_maxValue)
{
	m_value++; // m_value = m_value+1;
	m_view->OutData(m_value);
}
}
```

Виклик наступної функції призводить до зменшення значення лічильника на 1 за умови, що воно більше мінімально можливого.

```
void Counter_Model::Decrement()
{
if(m_value > m_minValue)
{
	m_value--;
```

```
        m_view->OutData(m_value);
}
}
```

Наступна функція дозволяє онулити значення лічильника. Якщо мінімально можливе значення лічильника не дорівнює нулю, тоді при виклику цієї функції встановлюється мінімально можливе значення лічильника.

```
void Counter_Model::Reset()
{
m_value = m_minValue;
m_view->OutData(m_value);
}
```

Ця функція дозволяє з’єднати клас моделі з класом представлення, за допомогою якого дані будуть відображені.

```
void Counter_Model::setView(View * view)
{
m_view = view;
}
```

Реалізація класу контролера. В цьому класі містяться два поля – вказівник на екземпляр класу драйвера порта вводу/виводу для можливості відстеження дій користувача щодо зміни значення лічильника шляхом натискання на кнопки, а також вказівник на екземпляр класу моделі. Контролер аналізує дії користувача і змінює модель. Таким чином, керування процесом зміни даних у відповідь на дії користувача, виконує саме цей клас.

```
class Counter_Controller
{
private:
Counter_Model * m_counterModel;
Port * m_buttonPort;

public:
Counter_Controller();
Counter_Controller(Counter_Model * model, Port * buttonPort);
void CheckUserAction();
};
```

В конструкторі за замовчуванням значення вказівників встановлюються в NULL.

```
Counter_Controller::Counter_Controller()
{
m_counterModel = NULL;
m_buttonPort = NULL;
}
```

В перевантаженому конструкторі передані в якості аргументів значення записуються в поля класу.

```
Counter_Controller::Counter_Controller(Counter_Model * counterModel, Port * buttonPort)
{
```

```
m_counterModel = counterModel;
m_buttonPort = buttonPort;
}
```

В наступній функції виконується перевірка дій користувача та зміна моделі відповідним чином.

```
void Counter_Controller::CheckUserAction()
{
if((m_counterModel != NULL) && (m_buttonPort != NULL))
{
	if(m_buttonPort->getPinState(0) == false) // 0b00000001
	{
		m_counterModel->Increment();
		_delay_ms(500);
	}

	if(m_buttonPort->getPinState(1) == false) // 0b00000010
	{
		m_counterModel->Decrement();
		_delay_ms(500);
	}

	if(m_buttonPort->getPinState(2) == false) // 0b00000100
	{
		m_counterModel->Reset();
		_delay_ms(500);
	}
}
}
```

Приклад використання розроблених класів представлено далі.

```
int main(void)
{
Port ledPort('C', OUTPUT);
Port buttonPort('B', INPUT);
View view(&ledPort);
Counter_Model counter(0, 0, 15);
counter.setView(&view);
Counter_Controller controller(&counter, &buttonPort);

      while (1)
      {
       controller.CheckUserAction();
}
}
```

Схема моделі двійкового лічильника в Proteus, за допомогою якої можна перевірити роботу програми, представлена на рис. 2 (файл Proteus_mode_Lection_3.pdsprj на гугл-диску додається):

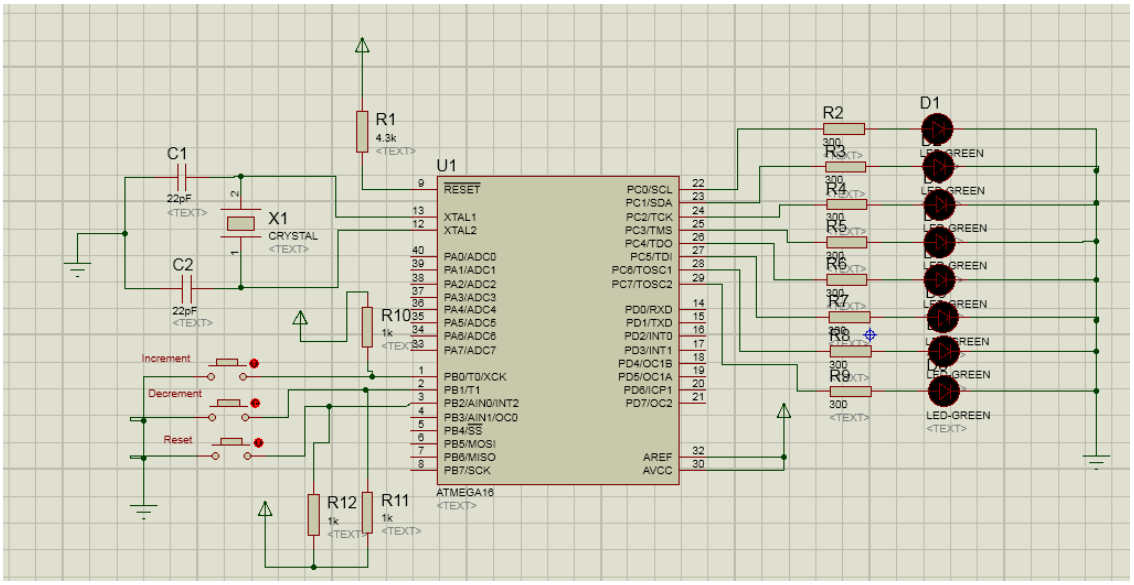


Рис. 2.

У випадку застосування такої моделі (паттерну MVC) при написанні програмного забезпечення для мікроконтролерів, розподіл задач виглядатиме наступним чином. Контролер відповідає за отримання та опрацювання сигналів від вхідних інтерфейсів (портів вводу/виводу). Для прикладу, коли, необхідно розрізнити натиснення кнопки, контролер здійснює цю задачу, будь-якими доступними в мікроконтролері засобами та передає сигнал в компонент (клас) Модель. Компонент Модель у випадку мікроконтролерів по суті реалізує поведінку пристрою в залежності від поточного стану та інформації отриманої з компоненту Контролер. Компонент Представлення у випадку мікроконтролерів – це пристрій візуалізації інформації (світлодіоди, семисегментний індикатор та ін.) чи певний виконавчий пристрій.

Завдання для самостійного виконання:

1. Прочитати «Розділ 11. Робота з класами» (стор. 535-590) в підручнику «Стивен Прата. Мова програмування C++ (6 видання)» (підручник на гугл-диску): https://drive.google.com/drive/u/0/folders/1MiXpe2uS_x0g-2UJGB89kGaOlhcDjzGi

2. Використовуючи розроблений в першому домашньому завданні клас для роботи з семисегментним індикатором розширити клас View, наведений в прикладі вище, для відображення значення лічильника імпульсів на багаторозрядному семисегментному індикаторі. Значення лічильника може бути в межах від 0 до 99. У звіті навести схему пристрою та код програми.

3. Розробити клас, що який дозволяє формувати на будь-якому виводі будь-якого порта мікроконтролера мікроконтролера ШІМ (широтно-імпульсна модуляція) сигнал. При цьому драйвер порта мікроконтролера має бути оформлений у вигляді іншого класу (як в прикладі з першого заняття). Скважність імпульсів задається знченням від 0 до 100. Частота ШІМ сигналу має бути 500 Гц. Розробити схему пристрою в Proteus та за допомогою віртуального осцилографа отримати осцилограми сигналу для значень скважносты 20, 50 та 80. В звіті навести код програми, принципову схему та отримані осцилограми.

4. Оформити звіт. Завантажити звіт на гугл-диск.

Практична робота 4.

Тема: аналого-цифрове перетворення сигналів з відображенням даних на екрані рідкокристалічного індикатора (розробка класів (драйверів) АЦП, алфавітно-цифрового рідкокристалічного індикатора, спрощеного класу (драйвера) портів мікроконтролера ATMega16).

Мікроконтролер ATMega16 містить 10-разрядний АЦП послідовного наближення. АЦП пов'язаний з 8-канальним аналоговим мультиплексором, 8 однополярних входів якого з'єднані з лініями порту А.

Робота АЦП дозволяється шляхом установки біта ADEN в ADCSRA. Вибір опорного джерела й каналу перетворення не можливо виконати до установки ADEN. Якщо ADEN = 0, то АЦП не споживає струм, тому, при переведенні в економічні режими сну рекомендується попередньо відключити АЦП.

АЦП генерує 10-розрядний результат, що міститься в парі регістрів даних АЦП ADCH й ADCL. За замовчуванням результат перетворення розташовується в молодших 10-ти розрядах 16-розрядного слова (вирівнювання праворуч), але може бути опціонально розміщений у старших 10-ти розрядах (вирівнювання ліворуч) шляхом установки біта ADLAR у регістрі ADMUX.

Регістри керування АЦП

Регістр керування мультиплексором АЦП – ADMUX

Розряд	7	6	5	4	3	2	1	0
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
Зчитування /запис	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.
Початкове значення	0	0	0	0	0	0	0	0

Розряд 7:6 – REFS1:0: Біти вибору джерела опорної напруги.
Дані біти визначають, яка напруга буде використовуватися в якості опорної для АЦП. Якщо змінити значення даних біт у процесі перетворення, то нові установки набудуть чинності тільки по завершенні поточного перетворення (тобто коли встановиться біт ADIF у регістрі ADCSRA). Внутрішнє джерело опорної напруги можна не використовувати, якщо до виводу AREF підключене зовнішнє опорне джерело.

REFS1	REFS0	Опорне джерело
0	0	AREF, внутрішнє джерело відключене
0	1	AVCC із зовнішнім конденсатором на виводі AREF
1	0	Зарезервовано
1	1	Внутрішнє джерело опорної напруги 2,56В с зовнішнім конденсатором на виводі AREF

Розряд 5 – ADLAR: Біт керування представленням результату перетворення.
Біт ADLAR впливає на представлення результату перетворення в парі регістрів результату перетворення АЦП. Якщо ADLAR = 1, то результат перетворення буде

вирівняний ліворуч, у противному випадку – праворуч. Дія біта ADLAR відбувається відразу після зміни, незалежно від перетворення, що виконується паралельно.

Розряд 4:0 – MUX4:0: Біти вибору аналогового каналу й коефіцієнта підсилення.

Дані біти визначають, які з наявних аналогових входів підключаються до АЦП. Крім того, з їхньою допомогою можна вибрати коефіцієнт підсилення для диференціальних каналів. Якщо значення біт змінити в процесі перетворення, то механізм їхньої дії набуде чинності тільки після завершення поточного перетворення (після установки біта ADIF у регістрі ADCSRA).

Регістр керування й статусу АЦП – ADCSRA

Розряд	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Зчитування /запис	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.
Початкове значення	0	0	0	0	0	0	0	0

Розряд 7 – ADEN: Дозвіл роботи АЦП.

Запис у даний біт лог. 1 дозволяє роботу АЦП. Якщо в даний біт записати лог 0, то АЦП відключається, навіть якщо він перебував у процесі перетворення.

Розряд 6 – ADSC: Запуск перетворення АЦП.

У режимі одиночного перетворення установка даного біта ініціює старт кожного перетворення. У режимі автоматичного перезапуску установкою цього біта ініціюється тільки перше перетворення, а всі інші виконуються автоматично. Перше перетворення після дозволу роботи АЦП, ініційоване бітом ADSC, виконується по розширеному алгоритму і триває 25 тактів синхронізації АЦП, замість звичайних 13 тактів. Це пов'язане з необхідністю ініціалізації АЦП.

У процесі перетворення при опитуванні біта ADSC повертається лог. 1, а по завершенні перетворення – лог. 0. Запис лог. 0 у даний біт не передбачене і не виконує ніякої дії.

Розряд 5 – ADFR: Вибір режиму автоматичного перезапуску АЦП.

Якщо в даний біт записати лог. 1, то АЦП перейде в режим автоматичного перезапуску. У цьому режимі АЦП автоматично виконує перетворення і модифікує регістри результату перетворення через фіксовані проміжки часу. Запис лог. 0 у цей біт припиняє роботу в даному режимі.

Розряд 4 – ADIF: Прапор переривання АЦП.

Даний прапор установлюється після завершення перетворення АЦП і відновлення регістрів даних. Якщо встановлені біти ADIE й I (регістр SREG), то відбувається переривання по завершенні перетворення. Прапор ADIF скидається апаратно при переході на відповідний вектор переривання. Альтернативно прапор ADIF скидається шляхом запису лог. 1 у нього. При виконанні команди "зчитування-модифікація-запис" з регістром ADCSRA очікуване переривання може бути відключено.

Розряд 3 – ADIE: Дозвіл переривання АЦП.

Після запису лог. 1 у цей біт, за умови, що встановлено біт I у регістрі SREG, дозволяється переривання по завершенні перетворення АЦП.

Розряди 2:0 – ADPS2:0: Біти керування дільником АЦП.
Дані біти визначають на яке значення тактова частота мікроконтролера буде відрізнятися від частоти вхідної синхронізації АЦП.

Регістри даних АЦП – ADCL й ADCH

ADLAR = 0:

Розряд	15	14	13	12	11	10	9	8
	-	-	-	-	-	-	ADC9	ADC8
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
	7	6	5	4	3	2	1	0
Зчитування /запис	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.
	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.
Початкове значення	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0

ADLAR = 1:

Розряд	15	14	13	12	11	10	9	8
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
	ADC1	ADC0	-	-	-	-	-	-
	7	6	5	4	3	2	1	0
Зчитування /запис	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.
	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.	Зчит.
Початкове значення	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0

По завершенні перетворення результат міститься в цих двох регістрах. При використанні диференціального режиму перетворення результат представляється в коді двійкового доповнення.

Якщо виконано зчитування ADCL, то доступ до цих регістрів для АЦП буде заблокований (тобто АЦП не зможе надалі модифікувати результат перетворення), поки не буде зчитаний регістр ADCH.

Лівосторонній формат представлення результату зручно використовувати, якщо достатньо 8 розрядів. У цьому випадку 8-розрядний результат зберігається в регістрі ADCH, і зчитування регістра ADCL можна не виконувати. При форматі вирівнювання праворуч необхідно спочатку зчитати ADCL, а потім ADCH.
ADC9:0: Результат перетворення АЦП.
Дані біти представляють результат перетворення.

Приклад 1. Програма виконує аналого-цифрове перетворення сигналу та виведення отриманого результату на світлодіодні індикатори у вигляді двійкового коду (приклад розробка класу (драйвера) для роботи з АЦП).

```
#define F_CPU 8000000

#define RS 0
#define RW 1
#define E 2
```

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

Власне клас, що реалізує функціональність драйера для роботи з АЦП, досить простий.

```
class MyADC
{
public:
MyADC();
int getValue(char channel);
};
```

В конструкторі за замовчуванням виконується встановлення біту дозволу роботи АЦП в регістрі керування, а також значення дільника частоти для того, щоб АЦП працював на коректній частоті.

```
MyADC::MyADC()
{
ADCSRA = 0b10000111;
}
```

Функція-член класу драйвера АЦП, що повертає значення виміряного сигналу, має вигляд:

```
int MyADC::getValue(char channel)
{
	char dl, dh;
	int result;

	ADMUX = 0b00000000 | channel;
	ADCSRA |= 0b01000000;
	while((ADCSRA & 0b00010000) == 0);

	dl = ADCL;
	dh = ADCH;
	result = dh;
	result = result << 8;
	result = result + dl;
	return result;
}

class MyPortC
{
public:
	MyPortC(bool isOutput);
	void outData(char value);
	void setBit(char pos);
	void resetBit(char pos);
};

MyPortC::MyPortC(bool isOutput)
{
if(isOutput == true)
{
	DDRC = 0b11111111;
}
}
```

```
void MyPortC::outData(char value)
{
	PORTC = value;
}

void MyPortC::setBit(char pos)
{
	PORTC|= (1<<pos);
}

void MyPortC::resetBit(char pos)
{
	PORTC&= ~(1<<pos);
}
```

Наступний код демонструє принципи використання класу драйвера для роботи з АЦП.

```
int main(void)
{
int value;

MyADC adc;
MyPortC portC(true);

while (1)
{
	value = adc.getValue(0);
	// ADC returns 10-bit value, but we have 8 LEDs, so the value is divided by 4
	portC.outData(value/4);
	_delay_ms(100);
}
}
```

На рис. 2 представлено схему моделі мікропроцесорної системи для перевірки роботи драйвера АЦП. Оцифроване значення відображається у двійковому вигляді на світлодіодах.

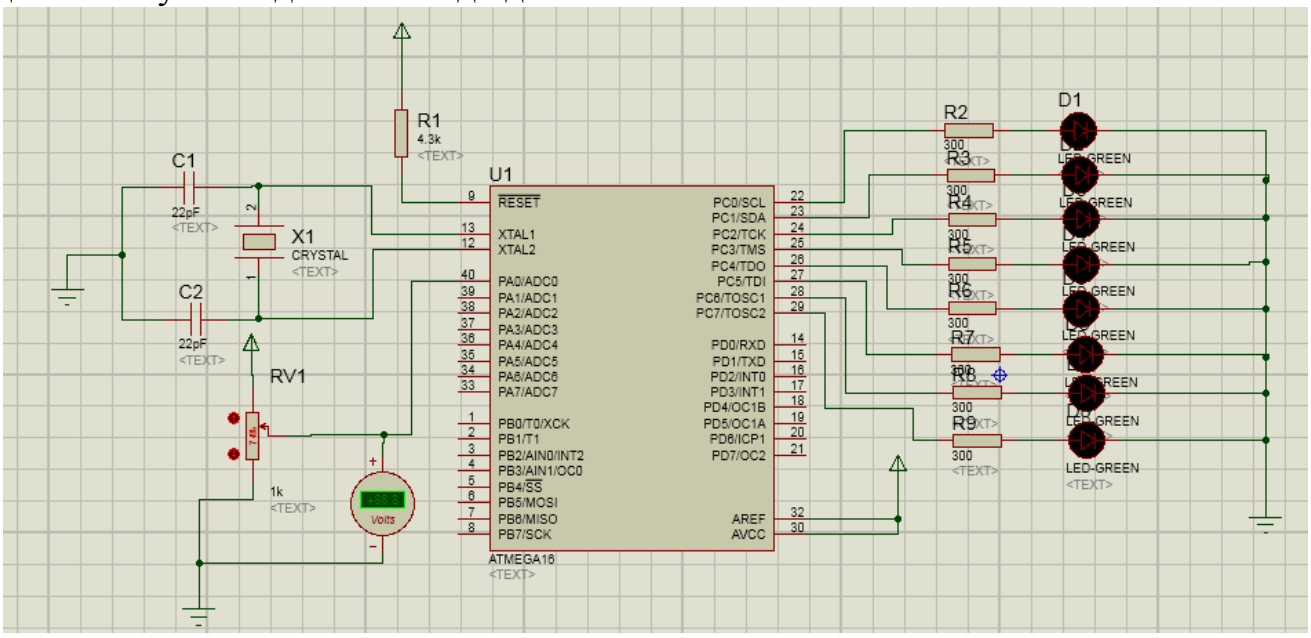


Рис. 1.

Алфавітно-цифрові РКІ-модулі є недорогим і зручним рішенням при розробці нових виробів, при цьому забезпечують відображення великого об'єму інформації при низькому енергоспоживанні.

Для підключення РКІ-модуля до управляючої системи використовується паралельна синхронна шина DB0...DB7, яка може бути 8-и або 4-х бітною (вибирається програмно), лінія вибору операції зчитування/запису R/W, лінія вибору регістра RS і лінія синхронізації E. Крім ліній управляючої шини є дві лінії для подачі напруги живлення 5В – GND і VCC, а також лінія для управління контрастністю – V_0.

Назви ліній шини є стандартними, але існує безліч різних варіантів розташування контактів в кожного конкретного конструктиву РКІ-модуля. Стандартним варіантом розташування контактів є дворядне 14-и контактне поле (табл. 4.3), розташоване вертикально в лівій частині модуля, а також співпадаюче з ним дворядне 16-ти контактне поле, що містить додаткову пару контактів з підключеними до неї виводами живлення для підсвічування. Для отримання достовірної інформації необхідно скористатися відповідною довідковою літературою виробника модуля.

Табл. 1.

N	Назва виводу	Опис
1	V_{SS}	(–) Живлення, 0 В.
2	V_{DD}	(+) Живлення, +5 В.
3	V_0	Напруга зміщення, якою регулюється контрастність
4	RS	Вхід. Високий рівень – дані, низький рівень – команди
5	R/W	Вхід. Високий рівень – читання, низький рівень – запис
6	E	Вхід. Строб, який супроводжує сигнали на шині "команда/дані"
7	DB_0	Шина даних (8-бітний режим)
8	DB_1	Шина даних (8-бітний режим)
9	DB_2	Шина даних (8-бітний режим)
10	DB_3	Шина даних (8-бітний режим)
11	DB_4	Шина даних (8-и і 4-х бітний режими, молодший біт)
12	DB_5	Шина даних (8-и і 4-х бітний режими)
13	DB_6	Шина даних (8-и і 4-х бітний режими)
14	DB_7	Шина даних (8-и і 4-х бітний режими, старший біт)

Послідовність дій, які необхідно виконувати управляючій системі при здійсненні операцій запису і зчитування для 8-ми розрядної шини демонструє часова діаграма, показана на рис. 2.

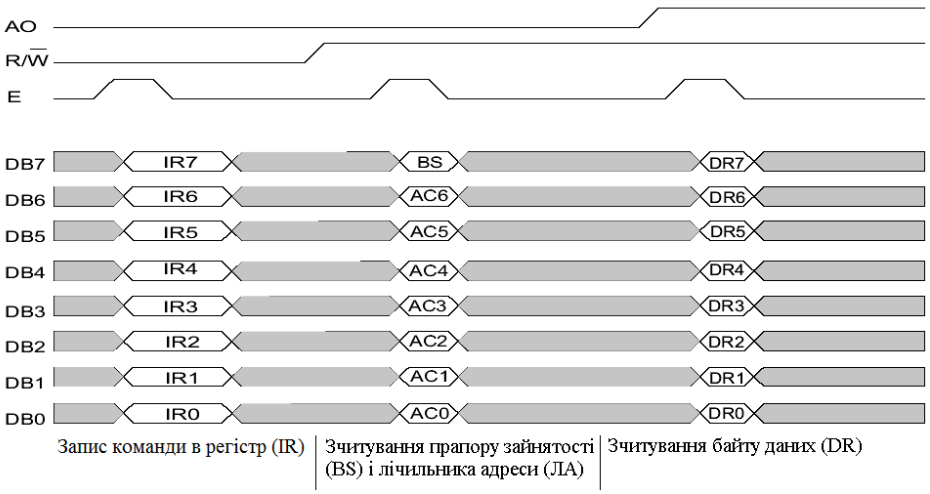


Рис. 2.

Операції запису для 8-ми разрядної шини полягають в наступному:
1. Встановити значення лінії RS.
2. Вивести значення байта даних на лінії шини DB0...DB7.
3. Встановити лінію E = 1.
4. Встановити лінію E = 0.
5. Встановити лінії шини DB0...DB7 = HI.

Для зчитування значення з модуля виконуються наступні операції:
1. Встановити значення лінії RS.
2. Встановити лінію R/W = 1.
3. Встановити лінію E = 1.
4. Зчитати значення байта даних з ліній шини DB0...DB7.
5. Встановити лінію E = 0.
6. Встановити лінію R/W = 0.

Програмне управління рідкокристалічним індикатором здійснюється за допомогою системи команд, що наведена в табл. 2.

Табл. 2.

Код										**Описання команди**	**Час виконання команди (f_{osc}= 250 кГц)**
RS	**R/W**	**DB7**	**DB6**	**DB5**	**DB4**	**DB3**	**DB2**	**DB1**	**DB0**		
0	0	0	0	0	0	0	0	0	1	Очистити дисплей і встановити курсор в нульовую позицію (адреса 0)	до 1,64 мс
0	0	0	0	0	0	0	0	1	*	Встановити курсор в нульовую позицію (адреса 0). Встановити дисплей відносно буфера DDRAM в початкову позицію. Вміст DDRAM при цьому не змінюється.	от 40 мкс до 1,6 мс

0	0	0	0	0	0	0	1	I/D	S	Встановити напрям зсуву курсора праворуч (I/D=1) або ліворуч (I/D=0) при записі/зчитуванні наступного коду в DDRAM. Дозволити (S=1) зсув дисплея разом із зсувом курсора.	40 мкс
0	0	0	0	0	0	1	D	C	B	Увімкнути(D=1)/вимкнути (D=0) дисплей. Запалити (C=1)/погасити (C=0) курсор. Зображення курсора зробити мігаючим (B=1).	40 мкс
0	0	0	0	0	1	S/C	R/L	*	*	Перемістити курсор (S/C=0) або зсунути дисплей (S/C=1) праворуч (R/L=1), або ліворуч (R/L=0).	40 мкс
0	0	0	0	1	DL	N	F	*	*	Встановити розрядність шини даних 4 біта (DL=0) або 8 біт (DL=1), кількість рядків дисплея - один (N=0) або два (N=1), шрифт - 5×7 точок (F=0) або 5×10 точок (F=1).	40 мкс
0	0	0	1	A_{CG}						Встановлення адреси CGRAM. Після цієї команди дані будуть записуватися/зчитуватись в/із CGRAM.	40 мкс
0	0	1	A_{DD}							Встановлення адреси DDRAM. Після цієї команди дані будуть записуватися/зчитуватись в/із DDRAM.	40 мкс
0	1	BF	AC							Зчитування стану busy-прапору (BF) та лічильника адреси.	1 мкс
1	0	Дані								Запис даних в DDRAM або CGRAM.	40 мкс
1	1	Дані								Зчитування даних з DDRAM или CGRAM.	40 мкс

DDRAM – Display data RAM – ОЗП ASCII-кодів символів, що відображаються
CGRAM – Character generator RAM – ОЗП знакогенератора

* – означає будь-яке значення

Модуль містить ОЗП для зберігання даних (DDRAM), що виводяться на екран. Розподіл адрес модуля представлено на рис. 3.

№ Знакомісця		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
АДРЕСА	1-й рядок	0h	1h	2h	3h	4h	5h	6h	7h	8h	9h	0Ah	0Bh	0Ch	0Dh	0Eh	0Fh
	2-й рядок	40h	41h	42h	43h	44h	45h	46h	47h	48h	49h	4Ah	4Bh	4Ch	4Dh	4Eh	4Fh

Рис. 3.

Відеопам'ять, що має загальний об'єм 80 байтів, призначена для зберігання кодів символів, що відображуються на РКІ. Відеопам'ять організована в два рядки по 40 символів в кожному. Ця прив'язка є жорсткою і не підлягає зміні.

Приклад 2. Програма виконує аналого-цифрове перетворення сигналу та виведення отриманого результату на алфавітно-цифровий рідкокристалічний індиатор.

```
#define F_CPU 8000000

#define RS 0
#define RW 1
#define E 2

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

class MyADC
{
public:
MyADC();
int getValue(char channel);

};

MyADC::MyADC()
{
ADCSRA = 0b10000111;
}

int MyADC::getValue(char channel)
{
char dl, dh;
int result;

ADMUX = 0b00000000 | channel;
ADCSRA |= 0b01000000;
while((ADCSRA & 0b00010000) == 0);

dl = ADCL;
dh = ADCH;
result = dh;
result = result << 8;
result = result + dl;
return result;
}

class MyPortC
{
public:
MyPortC(bool isOutput);
void outData(char value);
void setBit(char pos);
void resetBit(char pos);
};

MyPortC::MyPortC(bool isOutput)
{
if(isOutput == true)
{
	DDRC = 0b11111111;
}
}

void MyPortC::outData(char value)
```

```
{
PORTC = value;
}

void MyPortC::setBit(char pos)
{
PORTC|= (1<<pos);
}

void MyPortC::resetBit(char pos)
{
PORTC&= ~(1<<pos);
}


class MyPortD
{
public:
MyPortD(bool isOutput);
void outData(char value);
void setBit(char pos);
void resetBit(char pos);
};

MyPortD::MyPortD(bool isOutput)
{
if(isOutput == true)
{
	DDRD = 0b11111111;
}
}

void MyPortD::outData(char value)
{
	PORTD = value;
}

void MyPortD::setBit(char pos)
{
	PORTD|= (1<<pos);
}

void MyPortD::resetBit(char pos)
{
	PORTD&= ~(1<<pos);
}

class MyLCD
{
void write();
MyPortC *m_pDataPort;
MyPortD *m_pControlPort;

public:
MyLCD(MyPortC *dataPort, MyPortD *controlPort);
void clear();
void print(char *text);
void print(int value);
void printChar(char symbol);
};

MyLCD::MyLCD(MyPortC *dataPort, MyPortD *controlPort)
{
```

```
m_pDataPort = dataPort;
m_pControlPort = controlPort;

m_pControlPort->resetBit(RS);
m_pControlPort->resetBit(RW);

_delay_ms(1);

m_pDataPort->outData(0b00111000);
write();

m_pDataPort->outData(0b00001111);
write();

m_pDataPort->outData(0b00000110);
write();

m_pDataPort->outData(0b00000001);
write();
}


void MyLCD::write()
{
m_pControlPort ->setBit(E);
_delay_ms(1);
m_pControlPort -> resetBit(E);
_delay_ms(1);
}


void MyLCD::clear()
{
m_pControlPort -> resetBit(RS);
_delay_ms(1);

m_pDataPort -> outData(0b00000001);
write();
}

void MyLCD::printChar(char symbol)
{
m_pControlPort -> setBit(RS);
_delay_ms(1);

m_pDataPort -> outData(symbol);
write();
}

void MyLCD::print(char * text)
{
unsigned int i;

for(i=0; i < strlen(text); i++)
{
       printChar(text[i]);
}
}

void MyLCD::print(int value)
{
printChar(value / 1000 + '0');
printChar((value % 1000) / 100 + '0');
```

```
printChar(((value % 1000) % 100) / 10 + '0');
printChar(value % 10 + '0');
}

int main(void)
{
int value;

MyADC adc;
MyPortC portC(true);
MyPortD portD(true);
MyLCD myLCD(&portC, &portD);

while (1)
{
	value = adc.getValue(0);
	myLCD.clear();
	myLCD.print(value);
	_delay_ms(250);
}
}
```

На рис. 4 преставлено принципову схему системи вимірювання аналогового сигналу з відображенням даних на алфавітно-цифровому рідкокристалічному індикаторі.

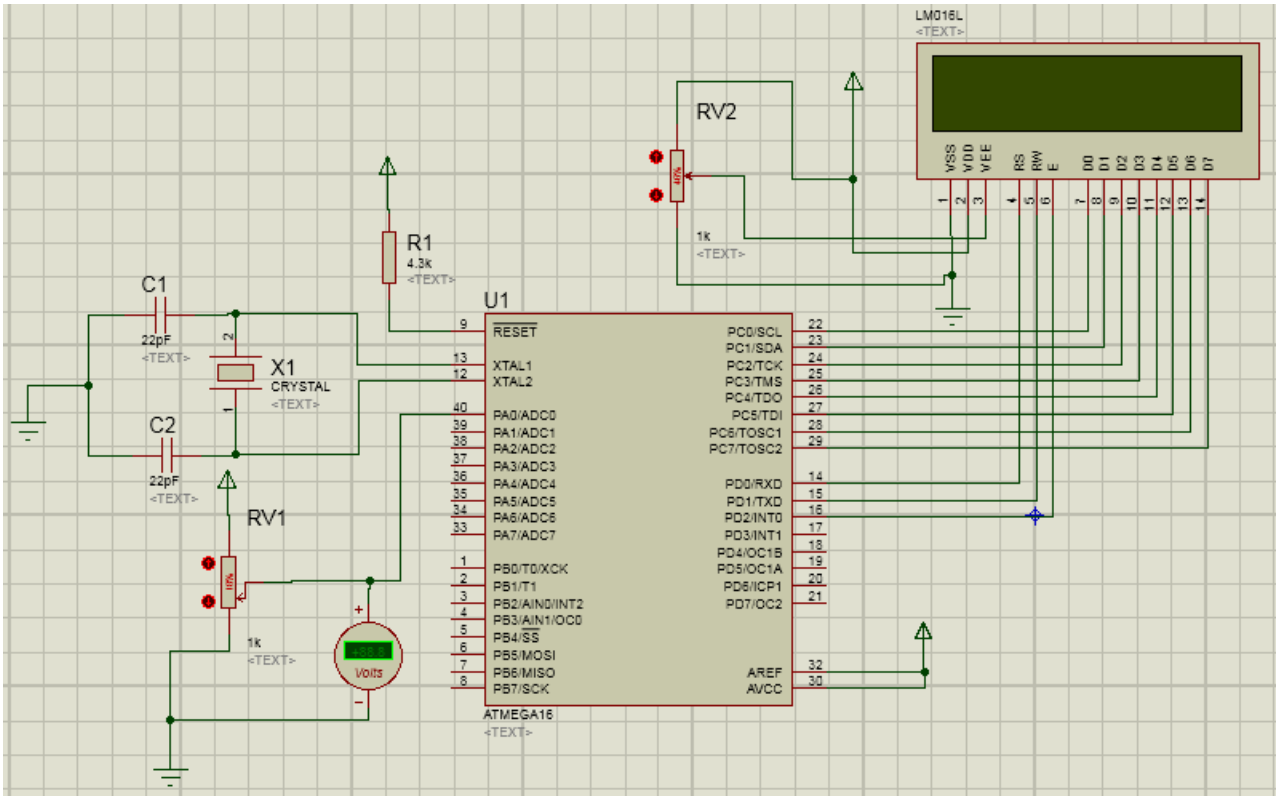


Рис. 4.

Завдання для самостійного виконання:

1. Прочитати «Розділ 10. Об’єкти і класи» (стор. 484-535) в підручнику «Стивен Прата. Мова програмування С++ (6 видання)» (підручник на гугл-диску):

https://drive.google.com/drive/u/0/folders/1MiXpe2uS_x0g-2UJGB89kGaOlhcDjzGi

2. Розширити наведений в прикладі клас для роботи з алфавітно-цифровим рідкокристалічним РКІ додавши нові в клас функції-члени для зміни положення курсору, режиму відображення курсору. Виконати перевантаження функції print() додавши до неї другий параметр – позицію на екрані РКІ, з якої потрібно виводити дані, або другий параметр має вказувати, на верхній або нижній рядок повинні виводитися дані.

3. Доробити програму таким чином, щоб на першому рядку РКІ відображались дані з каналу 0 АЦП, а на другому рядку РКІ відображались дані з каналу 1 АЦП (для цього також в схему потрібно додати ще один потенціометр, з'єднаний з каналом 1 АЦП мікроконтролера). Отримані значення потрібно відображати у вольтах у такому форматі:

U1: 3.726 V
U2: 1.935 V

Додаткову інформацію про АЦП можна знайти в розділі «3.4. Аналого-цифровий перетворювач» в підручнику «СХЕМОТЕХНІКА ЕЛЕКТРОННИХ ПРИСТРОЇВ ТА СИСТЕМ. ТОМ 3. Мікропроцесорна техніка» (файл є на гугл-диску), стор. 157-168.

4. Розробити клас «Годинник» (Clock). Написати програму, що демонструє роботу годинника. Відображати дані потрібно на екрані РКІ. Передбачити можливість налаштування часу (годин та хвилин) шляхом додавання в схему 4-х тактових кнопок – для збільшення/зменшення годин та збільщення/зменшення хвилин. Навести схему приладу. Інформація про алфавітно-цифровий РКІ та команди керування ним можна знайти в підручнику «СХЕМОТЕХНІКА ЕЛЕКТРОННИХ ПРИСТРОЇВ ТА СИСТЕМ. ТОМ 6. Апаратно-програмні засоби відображенні інформації» (файл є на гугл-диску), стор. 185-194.

Практична робота 5.

Тема: Принципи роботи з універсальним асинхронним приймачем/передавачем для передавання команд керування виконавчими механізмами (двигунами постійного струму). Реалізація паттерну проектування «Команда». Приклад програми – мікропроцесорна система керування мобільною роботизованою платформою (розробка класів (драйверів) порта мікроконтролера ATMega16, драйвера двигуна постійного струму, класу (драйвера) керування платформою, класу (драйвера) для роботи з універсальним асинхронним приймачем/передавачем USART):

На рис.1 зображено зовнішній вигляд мобільної роботизованої платформи.

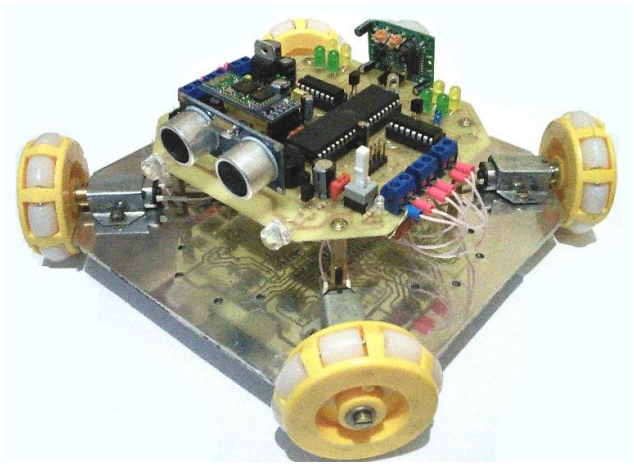


Рис. 1.

На роботизовану платформу встановлено «Шведське колесо» (Swedish wheel) або, як його ще називають, колесо Ілона, яке винайшов інженер зі шведської компанії Mecanum AB Бенгт Ілон у 1973 році – звідси й варіанти назв відповідно. Конструкція таких коліс (рис. 2) дозволяє обертатися на місці при мінімальній силі тертя та низькому обертальному моменті.

Рис. 2.

На рис. 3 представлені можливі напрямки руху платформи за різного напрямку обертання коліс.

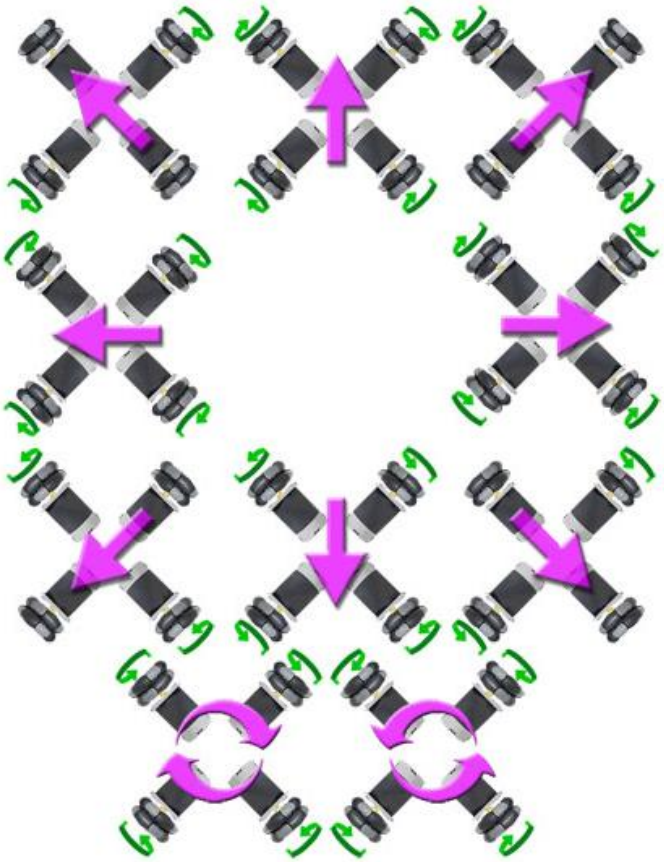


Рис. 3. Рух робота за різних напрямків обертання роликонесучих коліс

В якості виконавчих механізмів на платформі встановлено мотор-редуктори. Мотор-редуктор – це конструкція, яка утворюється, якщо з'єднати редуктор та електромотор (в даному випадку двигун постійного струму). Мініатюрний мотор-редуктор Pololy, показаний на рис. 4, використовується для різних робототехнічних проектів.

Рис. 4. Мотор-редуктор

Спрощена принципова схема моделі системи керування мобільною роботизованою платформою представлено на рис. 5.

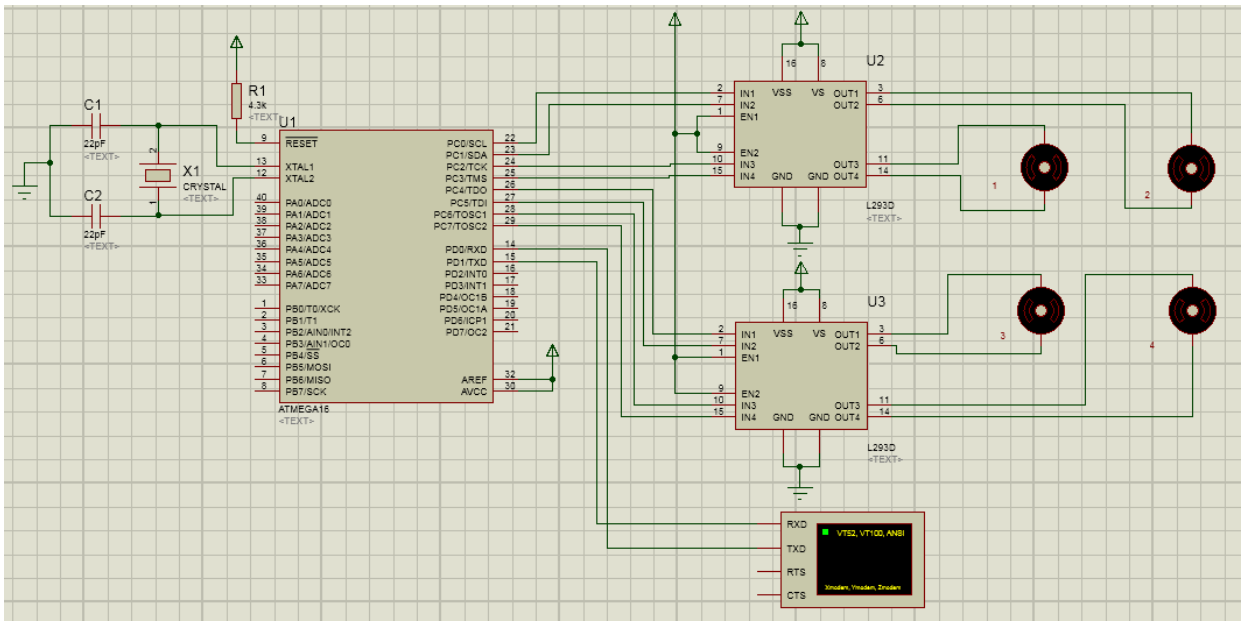


Рис. 5.

Для керування рухом мобільної роботизованої платформи використовується універсальний асинхронний приймач/передавач UART. Опис регістрів керування цим периферійним вузлом мікроконтролера наведено в Datasheet на мікроконтролер ATMega16 (файл MEGA16.pdf на Google диску), стор. 156-165. В моделі на рис. 5 для передавання команд керування роботизованою платформою по UART використовується віртуальний термінал. Однак в реальній системі для отримання команд встановлено модуль Bluetooth HC-05 (рис. 6).

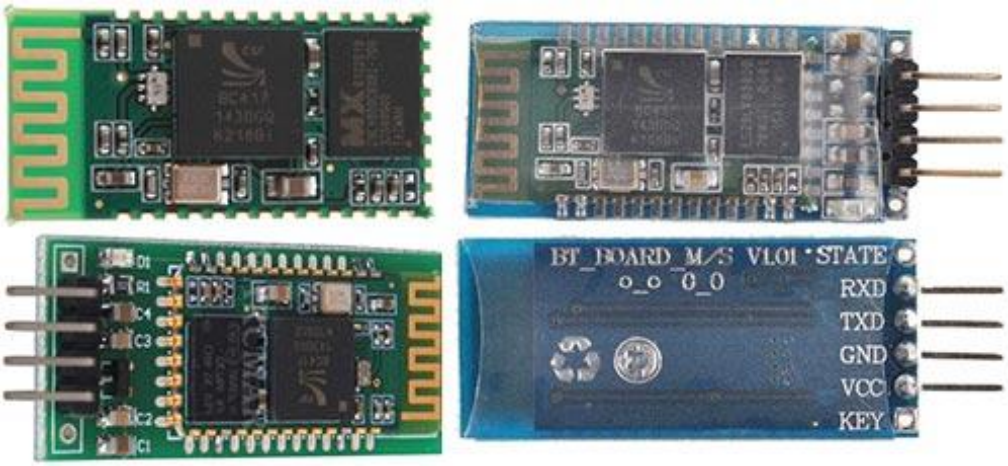


Рис. 6. Зовнішній вигляд Bluetooth-радіомодемів HC-05

Підключення цього модуля до мікроконтролера дуже просте – вивід TxD на модулі HC-05 з'єднується з виводом RxD мікроконтролера, а вивід RxD на модулі HC-05 з'єднується з виводом TxD мікроконтролера. Також на модуль потрібно подати живлення (таке саме, як у мікроконтролера: 3,3 В або 5 В), і не забути під'єднати вивід GND до загального проводу (землі). Модуль HC-05 забезпечує «прозоре» передавання команд, отриманих по Bluetooth, по інтерфейсу RS-232. Тобто, фактично цей модуль представляє собою перетворювач інтерфейсів Bluetooth –RS-232.

Приклад 1. Програма виконує зчитування даних, що надходять по універсальному асинхронному приймачу/передавачу USART (зокрема, отримні з Bluetooth-модуля HC-05, що підключений до входів USART мікроконтролера), і в залежності від отриманої команди мікроконтролер формує сигнали керування на двигуни постійного струму, що розташовані на мобільній роботизованій платформі.

Архітектуру програми представлено на рис. 6.

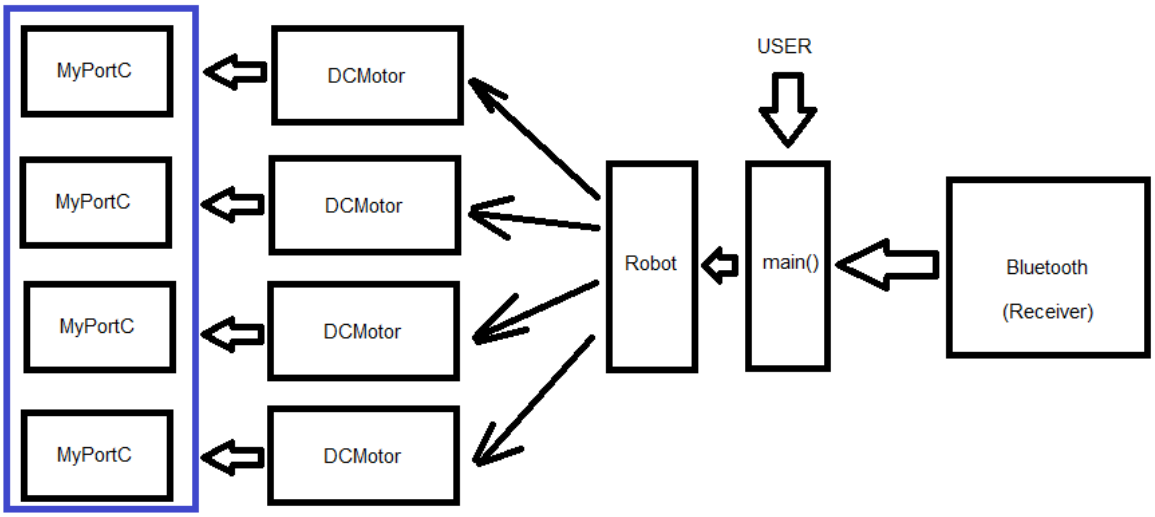


Рис. 6.

Текст програми представлено далі.

```
#define F_CPU 8000000

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

//////////////////Class PortC//////////////////////
class MyPortC
{
public:
MyPortC();
void setBit(char pos);
void resetBit(char pos);
};


MyPortC::MyPortC()
{
DDRC=0xFF;
}

void MyPortC::setBit(char pos)
{
if(pos>7) return;
PORTC |= (1<<pos);
}

void MyPortC::resetBit(char pos)
{
if(pos>7) return;
PORTC &= ~(1<<pos);
}
```

```
//////////////////DCMotor//////////////////
class DCMotor
{
MyPortC *m_port;
char m_pin1;
char m_pin2;
public:
DCMotor(MyPortC *port, char pin1, char pin2);
void Stop();
void RotateCW();
void RotateCCW();
};

DCMotor::DCMotor(MyPortC *port, char pin1, char pin2)
{
m_port = port;
m_pin1 = pin1;
m_pin2 = pin2;
}

void DCMotor::Stop()
{
m_port->resetBit(m_pin1);
m_port->resetBit(m_pin2);
}

void DCMotor::RotateCW()
{
m_port->setBit(m_pin1);
m_port->resetBit(m_pin2);
}

void DCMotor::RotateCCW()
{
m_port->resetBit(m_pin1);
m_port->setBit(m_pin2);
}

//////////////Robot///////////////////////
class Robot
{
DCMotor *m_dc1;
DCMotor *m_dc2;
DCMotor *m_dc3;
DCMotor *m_dc4;
public:
Robot(DCMotor *dc1, DCMotor *dc2, DCMotor *dc3, DCMotor *dc4);
void Stop();
void MoveNorth();
void MoveSouth();
void MoveWest();
void MoveEast();
};

Robot::Robot(DCMotor *dc1, DCMotor *dc2, DCMotor *dc3, DCMotor *dc4)
{
m_dc1 = dc1;
m_dc2 = dc2;
m_dc3 = dc3;
m_dc4 = dc4;
}
```

```
void Robot::Stop()
{
m_dc1->Stop();
m_dc2->Stop();
m_dc3->Stop();
m_dc4->Stop();
}

void Robot::MoveNorth()
{
m_dc1->RotateCCW();
m_dc2->RotateCW();
m_dc3->RotateCCW();
m_dc4->RotateCW();
}

void Robot::MoveSouth()
{
m_dc1->RotateCW();
m_dc2->RotateCCW();
m_dc3->RotateCW();
m_dc4->RotateCCW();
}

void Robot::MoveWest()
{
m_dc1->RotateCW();
m_dc2->RotateCW();
m_dc3->RotateCCW();
m_dc4->RotateCCW();
}

void Robot::MoveEast()
{
m_dc1->RotateCCW();
m_dc2->RotateCCW();
m_dc3->RotateCW();
m_dc4->RotateCW();
}
```

Клас драйвера досить простий. Фактично, в даному випадку немає потреби додавати в клас будь-які поля, а використовувати значення за замовчуванням. З одного боку це спрощує реалізацію класу, але з іншого боку, обмежує діапазон налаштувань, що можуть бути використані при організації зв’язку, значеннями за замовчуванням.

```
//Bluetooth
class Bluetooth
{
public:
Bluetooth();
char receiveChar();
};
```

В конструкторі за замовсуванням виконується налаштування універсального асинхронного приймача/передавача шляхом запису відповідних значень в регістри керування. Встановюється такий протокол передавання даних: 8 біт даних, 1 стоповий біт, перевірка на парність відсутя, швидкість передавання 9600 біт/с.

```
Bluetooth::Bluetooth()
{
 UCSRB = 0b00011000;
 UCSRC = 0b00000110;
 UBRRH = 0;
 UBRRL = 51;
}
```

Функція для отримання прийнятого символу.

```
char Bluetooth::receiveChar()
{
char c;
while((UCSRA & (1<<RXC)) == 0);
c = UDR;
return c;
}
```

Клас драйвера може бути розширений шляхом додавання функцій-членів класу, що виконують відправку одного символу, відправку рядка, або декількох байт та інші.

```
int main(void)
{
char cmd;
MyPortC port;
DCMotor dc1(&port, 2, 3);
DCMotor dc2(&port, 0, 1);
DCMotor dc3(&port, 4, 5);
DCMotor dc4(&port, 6, 7);
Robot robot(&dc1, &dc2, &dc3, &dc4);
Bluetooth bluetooth;

while(1)
{
	cmd = bluetooth.receiveChar();

	if(cmd == 'S')
	{
		robot.Stop();
	}
	else if(cmd == 'n')
	{
		robot.MoveNorth();
	}
	else if(cmd == 's')
	{
		robot.MoveSouth();
	}
	else if(cmd == 'w')
	{
		robot.MoveWest();
	}
	else if(cmd == 'e')
	{
		robot.MoveEast();
	}

}
}
```

В наведеному програмному коді можна бачити, що клас системи керування роботом пов’язаний з класом-драйвером двигуна постіного струму відношенням асоціації. При надходженні команди по UART (Bluetooth) відбувається виклик відповідного методу класу робота, в якому, в свою чергу, за допомогою вказівників на об’єкти, що керують двигунами постіного струму, виконується виклик методів для задання потрібного напрямку руху платформи. Тобто, один із об'єктів інтерфейсу (в даному прикладі вікно терміналу) безпосередньо викликає метод одного з об'єктів бізнес-логіки (робота), передаючи за необхідності (але не в даному випадку) до нього якісь параметри. З одного боку архітектура програми досить зрозуміла та очевидна. З іншого боку, якщо потрібно додати ще команди, наприклад обертання платформи за годинниковою або проти годинникової стрілки, тоді зміни потрібно вносити саме в клас системи керування роботом Robot, адже саме в цьому класі міститься вся логіка керування. І це не досить вдалий підхід, адже випадково може бути порушена існуюча функціональність. Як один із способів вирішення такого завдання є використання паттерну Команда (Command).

Команда – це поведінковий патерн проектування, який перетворює запити на об'єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій. Хороші програми зазвичай структуровані у вигляді кількох шарів. Паттерн Команда пропонує більше не надсилати запитивід одного об’єкта до іншого безпосередньо. Натомість кожен виклик, який відрізняється від інших, слід завернути у власний клас з єдиним методом, який і здійснюватиме виклик. Такі об'єкти називаються командами.

Класи команд можна об'єднати під загальним інтерфейсом з єдиним способом запуску. Після цього ті самі відправники зможуть працювати з різними командами, не прив'язуючись до їхніх класів. Команди можна буде взаємозамінювати на льоту, змінюючи підсумкову поведінку відправників. Параметри, з якими повинен бути викликаний метод об'єкта одержувача, можна заздалегідь зберегти в полях об'єкта-команди. Завдяки цьому об'єкти, що надсилають запити, можуть не турбуватися про те, щоб зібрати необхідні для одержувача дані. Більше того, тепер вони взагалі не знають, хто буде одержувачем запиту. Вся ця інформація прихована всередині команди. Структура паттерна Команда представлена на рис. 7.

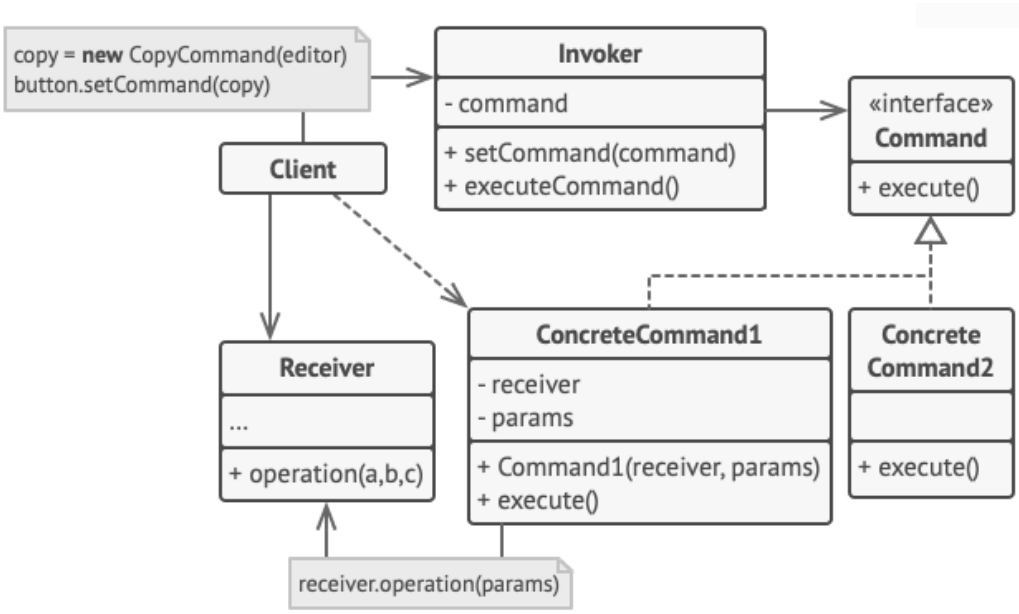


Рис. 7.

Відправник зберігає посилання на об'єкт команди та звертається до нього, коли потрібно виконати якусь дію. Відправник працює з командами лише через їхній спільний інтерфейс. Він знає, яку конкретно команду використовує, оскільки отримує готовий об'єкт команди від клієнта.

Команда описує загальний для всіх конкретних команд інтерфейс. Зазвичай тут описано лише один метод для запуску команди. Конкретні команди реалізують різні запити, дотримуючись загального інтерфейсу команд. Зазвичай команда робить всю роботу самостійно, лише передає виклик одержувачу, яким є однин з об'єктів бізнес-логіки.

Параметри, з якими команда звертається до одержувача, слід зберігати як поля. Найчастіше об'єкти команд можна зробити незмінними, передаючи у них всі необхідні параметри лише через конструктор.

Отримувач містить бізнес-логіку програми. У цій ролі може бути практично будь-який об'єкт. Зазвичай, команди перенаправляють виклики одержувачам. Але іноді, щоб спростити програму, можна позбавитися від одержувачів, розмістивши їх код у класи команд.

Клієнт створює об'єкти конкретних команд, передаючи в них усі необхідні параметри, серед яких можуть бути посилання на об'єкти одержувачів. Після цього клієнт пов'язує об'єкти відправників із створеними командами.

Код програми для керування рухом мобільної роботизованої платформи, в якому використано патерн Команда, представлено далі.

```
#define F_CPU 8000000

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

class PortC
{
public:
```

```
        PortC();
        void setBit(char pos);
        void resetBit(char pos);
};

PortC::PortC()
{
        DDRC = 0b11111111;
}

void PortC::setBit(char pos)
{
        PORTC |= (1 << pos);
}

void PortC::resetBit(char pos)
{
        PORTC &= ~(1 << pos);
}


class DCMotor
{
private:
        PortC * m_port;
        char m_pin1;
        char m_pin2;

public:
        DCMotor(PortC * port, char pin1, char pin2);
        void Stop();
        void RotateCW(); // обертання за годинниковою стрілкою (CW - clockwise)
        void RotateCCW();  // обертання проти годинникової стрілки (CCW - counter clockwise)
};


DCMotor::DCMotor(PortC * port, char pin1, char pin2)
{
        m_port = port;
        m_pin1 = pin1;
        m_pin2 = pin2;
}


void DCMotor::Stop()
{
        m_port->resetBit(m_pin1);
        m_port->resetBit(m_pin2);
}

void DCMotor::RotateCW()
{
        m_port->setBit(m_pin1);
        m_port->resetBit(m_pin2);
}

void DCMotor::RotateCCW()
{
        m_port->resetBit(m_pin1);
        m_port->setBit(m_pin2);
}


class Bluetooth
{
public:
```

```
        Bluetooth();
        char recieveChar();
};

Bluetooth::Bluetooth()
{
        UCSRA = 0;
        UCSRB = (1 << RXEN) | (1 << TXEN);
        UCSRC = 0b00000110;

        UBRRH = 0;
        UBRRL = 51;
}

char Bluetooth::recieveChar()
{
        char data;

        while((UCSRA & (1 << RXC)) == 0); //0b10000000
        data = UDR;
        return data;
}


class Command
{
protected:
        DCMotor* m_pMotor1;
        DCMotor* m_pMotor2;
        DCMotor* m_pMotor3;
        DCMotor* m_pMotor4;
public:
        Command(DCMotor* pM1, DCMotor* pM2, DCMotor* pM3, DCMotor* pM4)
        {
                m_pMotor1 = pM1;
                m_pMotor2 = pM2;
                m_pMotor3 = pM3;
                m_pMotor4 = pM4;
        }

        virtual void execute()
        {
                // Dummy
        }
};

class StopCommand : public Command
{
public:
        StopCommand(DCMotor* pM1, DCMotor* pM2, DCMotor* pM3, DCMotor* pM4) : Command(pM1, pM2,
pM3, pM4){};
        void execute()
        {
                        m_pMotor1->Stop();
                        m_pMotor2->Stop();
                        m_pMotor3->Stop();
                        m_pMotor4->Stop();
        }
};

class MoveNorthCommand : public Command
{
        public:
        MoveNorthCommand(DCMotor* pM1, DCMotor* pM2, DCMotor* pM3, DCMotor* pM4) : Command(pM1,
pM2, pM3, pM4){};
        void execute()
```

```
	{
		m_pMotor1->RotateCW();
		m_pMotor2->RotateCW();
		m_pMotor3->RotateCCW();
		m_pMotor4->RotateCCW();
	}
};

class MoveSouthCommand : public Command
{
	public:
	MoveSouthCommand(DCMotor* pM1, DCMotor* pM2, DCMotor* pM3, DCMotor* pM4) : Command(pM1,
pM2, pM3, pM4){};
	void execute()
	{
		m_pMotor1->RotateCCW();
		m_pMotor2->RotateCCW();
		m_pMotor3->RotateCW();
		m_pMotor4->RotateCW();
	}
};


class MoveWestCommand : public Command
{
	public:
	MoveWestCommand(DCMotor* pM1, DCMotor* pM2, DCMotor* pM3, DCMotor* pM4) : Command(pM1,
pM2, pM3, pM4){};
	void execute()
	{
		m_pMotor1->RotateCW();
		m_pMotor2->RotateCW();
		m_pMotor3->RotateCW();
		m_pMotor4->RotateCW();
	}
};

class MoveEastCommand : public Command
{
	public:
	MoveEastCommand(DCMotor* pM1, DCMotor* pM2, DCMotor* pM3, DCMotor* pM4) : Command(pM1,
pM2, pM3, pM4){};
	void execute()
	{
		m_pMotor1->RotateCCW();
		m_pMotor2->RotateCCW();
		m_pMotor3->RotateCCW();
		m_pMotor4->RotateCCW();
	}
};


class Robot
{
	Command * m_pCommand;
public:
	Robot();
	void SetCommand(Command * pCmd);
	void ExecuteCommand();
};

Robot::Robot()
{
	m_pCommand = NULL;
}
```

```
void Robot::SetCommand(Command * pCmd)
{
	m_pCommand = pCmd;
}

void Robot::ExecuteCommand()
{
	if(m_pCommand != NULL)
	{
		m_pCommand->execute();
	}
}

int main(void)
{
   PortC myPort;
   DCMotor motor1(&myPort, 2, 3);
   DCMotor motor2(&myPort, 0, 1);
   DCMotor motor3(&myPort, 4, 5);
   DCMotor motor4(&myPort, 6, 7);

   StopCommand stopCmd(&motor1, &motor2, &motor3, &motor4);
   MoveNorthCommand moveNorthCmd(&motor1, &motor2, &motor3, &motor4);
   MoveSouthCommand moveSouthCmd(&motor1, &motor2, &motor3, &motor4);
   MoveEastCommand moveEastCmd(&motor1, &motor2, &motor3, &motor4);
   MoveWestCommand moveWestCmd(&motor1, &motor2, &motor3, &motor4);

   Robot myRobot;
   Bluetooth receiver;

   char cmd;

    while (1)
    {
		cmd = receiver.recieveChar();

		if(cmd == 'n')
		{
			myRobot.SetCommand(&moveNorthCmd);
		}
		else if(cmd == 's')
		{
			myRobot.SetCommand(&moveSouthCmd);
		}
		else if(cmd == 'e')
		{
			myRobot.SetCommand(&moveEastCmd);
		}
		else if(cmd == 'w')
		{
			myRobot.SetCommand(&moveWestCmd);
		}
		else if(cmd == 'S')
		{
			myRobot.SetCommand(&stopCmd);
		}

		myRobot.ExecuteCommand();

    }
}
```

Після запуску процесу моделювання у вікні віртуального терміналу можна ввести одну з команд (n, w, s, e, S), та побачити, як змінюється напрям обертання двигунів постійного струму.

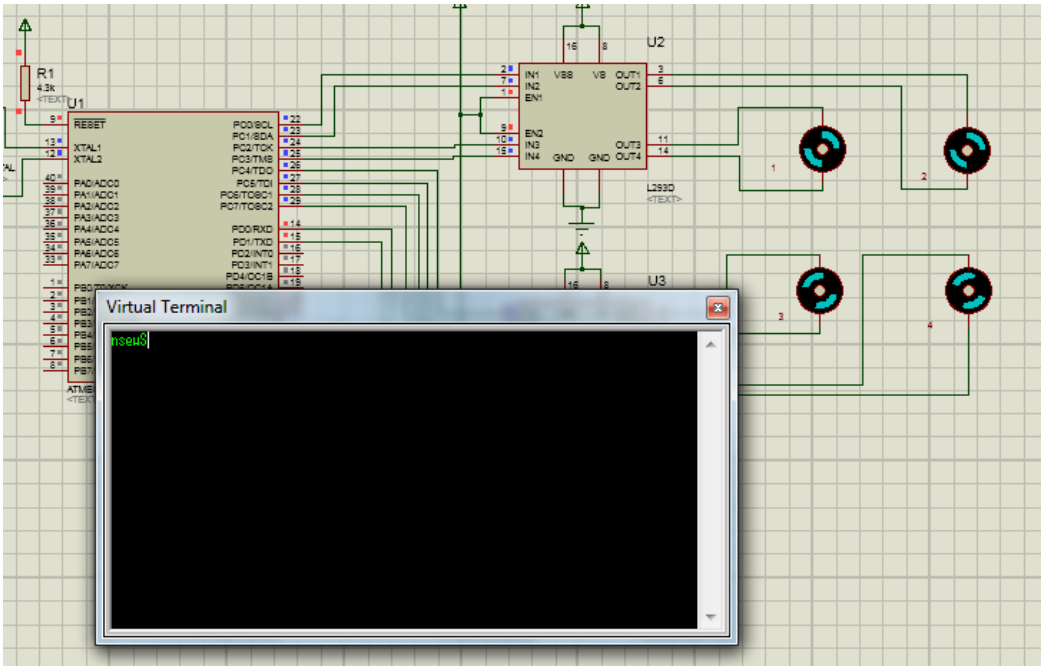


Рис. 8.

Паттерн Команда перетворює операції на об'єкти. А об'єкти можна передавати, зберігати та взаємозамінювати всередині інших об'єктів. Отже, для реалізації цього паттерну потрібно створити спільний інтерфейс команд і визначити метод запуску. Після цього один за одним створити класи певних команд. У кожному класі має бути поле для зберігання посилання на один або декілька об'єктів-отримувачів, яким команда перенаправлятиме основну роботу. Крім цього, команда повинна мати поля для зберігання параметрів, які потрібні для виклику методів одержувача. Значення всіх цих полів команда має набувати через конструктор. І, нарешті, потрібно реалізувати основний метод команди, викликаючи у ньому ті чи інші методи одержувача.

Завдання для самостійного виконання:

1. Доповнити клас Robot, що наведений у прикладі програми, функціями, що забезпечують рух платформи по діагоналі (4 функції) та обертання навколо вісі за та проти годинникової стрілки (2 функції): MoveSouthWest(), MoveSouthEast(), MoveNorthWest(), MoveNorthEast(), MoveCW(), MoveCCW().

2. Доповнити принципову схему мікрокпроцесорної системи керування мобільною роботизованою платформою шляхом підключення до мікроконтролера алфавітно-цифрового рідкокристалічного індикатора. Схема моделі в Proteus розміщено на Google-диску (файл Proteus_model_Lection_5.pdsprj).

3. Доробити програму таким чином, щоб на екрані РКІ відображався поточний напрям руху роботизованої платформи. Наприклад, якщо було надіслано команду ‘n’, то платформа рухатиметься на північ (вперед), і на екрані РКІ має відображатися слово «NORTH», а якщо було надіслано команду ‘w’, то

платформа рухатиметься на захід (ліворуч), і на екрані РКІ має відображатися слово «WEST». Для реалізації цього завдання потрібно розробити драйвер (клас) для роботи з портами та алфавітно-цифровим РКІ. Для шини даних РКІ пропонується використовувати порт B мікроконтролера, а для сигналів керування – виводи 5, 6 та 7 порта D мікроконтролера. Інформація про алфавітно-цифровий РКІ та команди керування ним можна знайти в підручнику «СХЕМОТЕХНІКА ЕЛЕКТРОННИХ ПРИСТРОЇВ ТА СИСТЕМ. ТОМ 6. Апаратно-програмні засоби відображенні інформації» (файл є на гугл-диску), стор. 185-194.

4. Додати два класи-команди (паттерн Команда) для реалізації обертання платформи на місці за годинниковою стрілкою та проти годинникової стрілки. Для цього по UART (Bluetooth) мають бути отримані відповідно символи ‘c’ та ‘r’.

5. Оформити звіт, в якому навести розроблені програми та схеми пристроїв. Завантажити звіт на гугл-диск.

Практична робота 6.

Тема: Принципи використання таймерів/лічильників при вирішенні завдань вимірювання характеристик дискретних сигналів (тривалості імпульсу, періоду сигналу). Приклад програми – мікропроцесорна система вимірювання відстані до перешкоди (розробка класів (драйверів) порта мікроконтролера ATMega16, драйвера ультразвукового датчика відстані HC-SR04, класу (драйвера) для роботи з універсальним асинхронним приймачем/передавачем USART).

Ультразвуковий датчик HC-SR04, представлений на рис. 1, може служити датчиком відстані для робота, завдяки якому він зможе визначати дистанцію до об'єктів, об'їжджати перешкоди, або будувати карту приміщення. Його можна також використовувати в якості датчика для сигналізації, що спрацьовує при наближенні об'єктів.

Рис. 1.

Ультразвуковий датчик визначає відстань до об'єктів таким самим чином, як це роблять дельфіни або кажани. Він генерує звукові імпульси на частоті 40 кГц і приймає відлуння, як показано на рис. 2. За часом поширення звукової хвилі туди і назад можна однозначно визначити відстань до об'єкта.

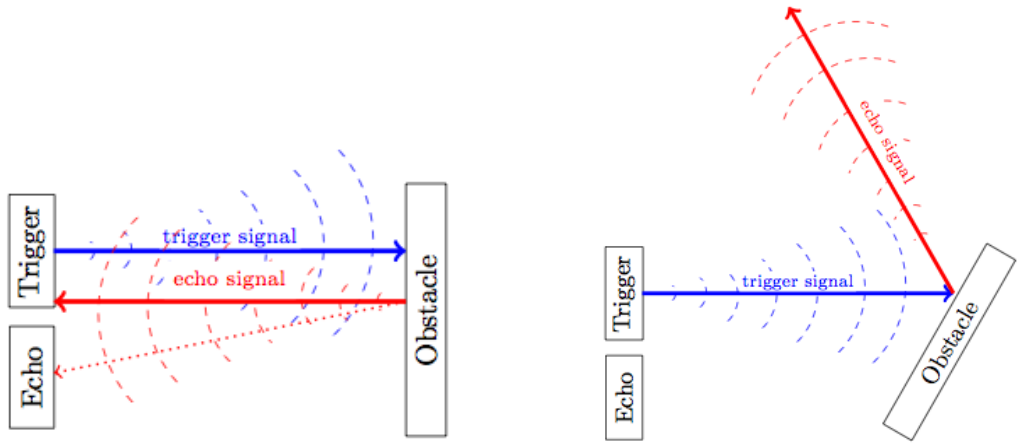


Рис. 2.

На відміну від інфрачервоних датчиків відстані, на покази ультразвукового далекоміра не впливають світлове випромінювання від сонця або колір об'єкта. Але можуть виникнути труднощі з визначенням відстані до пухнастих або дуже тонких предметів.

Опис виводів:

– Vcc – контакт живлення.

– Trig – цифровий вхід. Для запуску вимірювання необхідно подати на цей вхід логічну одиницю на 10 мкс. Наступне вимірювання рекомендується виконувати не раніше ніж через 50 мс.

– Echo – цифровий вихід. Після завершення вимірювання, на цей вихід буде подана логічна одиниця на час, пропорційний відстані до об’єкта.

– GND – від’ємний контакт живлення.

Роботу з датчиком можна умовно розділити на 4 етапи:

1. Подача імпульсу тривалістю 10 мкс на вхід Trig (рис. 3).

2. Всередені датчика вхідний імпульс перетворюється в 8 імпульсів частотою 40 кГц і відправляється вперед через випромінювач.

3. Відправлені імпульси відбиваються від перешкоди і приймаються приймачем. Отримується вихідний сигнал на виводі Echo.

4. Безпосередньо на стороні мікроконтролера отриманий сигнал перетворюється у відстань по формулі:

тривалість імпульсу(мкс)/58 = відстань (см),

тривалість імпульсу(мкс)/148 = відстань(дюйм).

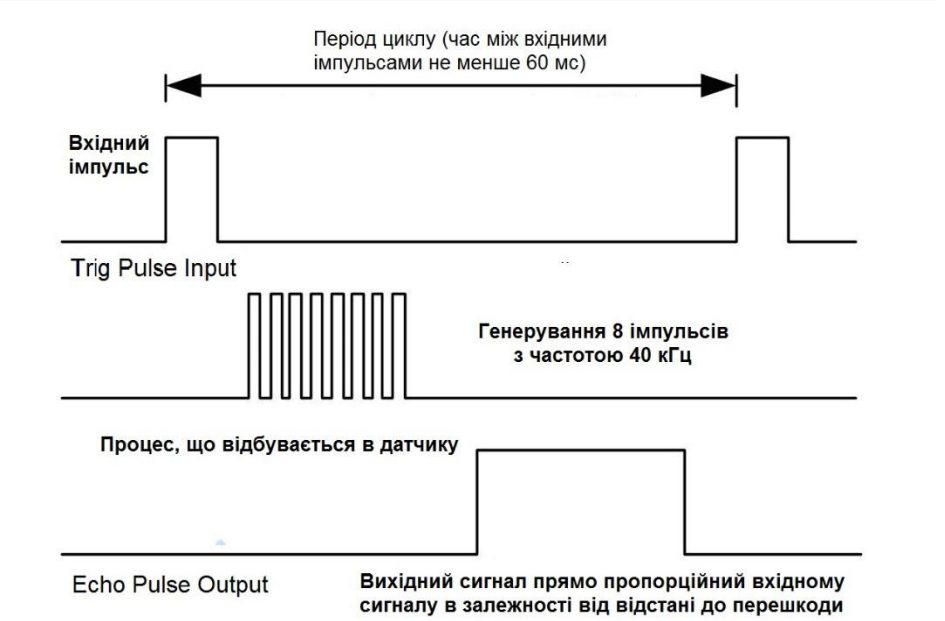


Рис. 3.

Для вимірювання тривалості імпульса з високою точністю зручно використовувати таймер-лічильник. 16-розрядні таймери-лічильники призначені для точного задання часових інтервалів, генерації прямокутних імпульсів і виміру часових характеристик імпульсних сигналів. Вміст 16-розрядного лічильника розбито на два 8-розрядних регістри, розташованих у пам'яті вводу-виводу: старший байт лічильника (TCNTn), у якому зберігаються старші 8 розрядів лічильника, і молодший байт лічильника (TCNTn), у якому зберігаються молодші 8 розрядів.

Таймер-лічильник 1 – TCNT1H й TCNT1L

Розряд	7	6	5	4	3	2	1	0
	TCNT1[15:8]							
	TCNT1[7:0]							
Зчитування /запис	Зч. /Зап	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.
Початкове значення	0	0	0	0	0	0	0	0

Два регістри в області вводу-виводу (TCNTn й TCNTn, разом TCNTn) дають повний доступ, як на зчитування, так і на запис до 16-розрядного лічильника.
Керування запуском/зупинкою таймера відбувається за допомогою регістра TCCR1B.

Регістр B керування таймером-лічильником 1 – TCCR1B

Розряд	7	6	5	4	3	2	1	0
	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
Зчитування /запис	Зч. /Зап	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.
Початкове значення	0	0	0	0	0	0	0	0

Розряд 7 – ICNCn: Фільтр шуму на вході захоплення.
Установка даного біта (запис лог. 1) активізує фільтр шуму на вході захвата. Логіка роботи фільтра складається у визначенні чотирьох підряд рівних за значенням вибірок і тільки в цьому випадку зміни свого вихідного стану. Отже, після дозволу придушення шумів сигнал із входу захопленя буде затримуватися на 4 такти системної синхронізації.
Розряд 6 – ICESn: Вибір типу фронту на вході захвата.
Даний біт дозволяє задати, який фронт на вході захвату ICPn призведе до захоплення стану таймера. Якщо ICESn=0, то зріз сигналу призводить до захоплення стану таймера, а якщо ж ICESn=1, то фронт призводить до виникнення захоплення.
Якщо відповідно до установки ICESn виникає умова захоплення, то вміст лічильника копіюється в регістр захвату ICRn. При цьому також встановлюється прапор захоплення ICFn, що може використовуватися для генерації переривання по захопленню (якщо дане переривання дозволене).
Якщо регістр ICRn використовується для зберігання значення верхньої межі рахунку, то вхід ICPn відключається від відповідного виводу мікроконтролера і функція захоплення блокується.
Розряд 5 – Зарезервований біт.
Даний біт зарезервований для подальшого використання. З метою сумісності з майбутніми розробками рекомендується під час запису в регістр TCCRn у даному розряді вказувати лог. 0.
Розряд 4:3 – WGMn3:2: Режим роботи таймера-лічильника.
Розряд 2:0 – CSn2:0: Вибір тактового джерела (стор. 107 в Datasheet).

Приклад 1. Програма виконує зчитування даних з ультразвукового датчика выдстані і надсилає отримані значення в текстовому форматі через універсальний асинхронний приймач/передавач кожні 500 мс.

Архітектура програми представлено на рис. 4.

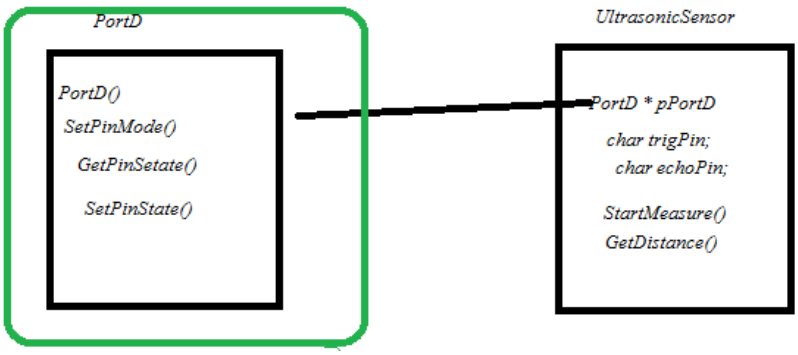


Рис. 4.

Схему мікропроцесорної системи вимірювання відстані до перешкоди представлено на рис.5.

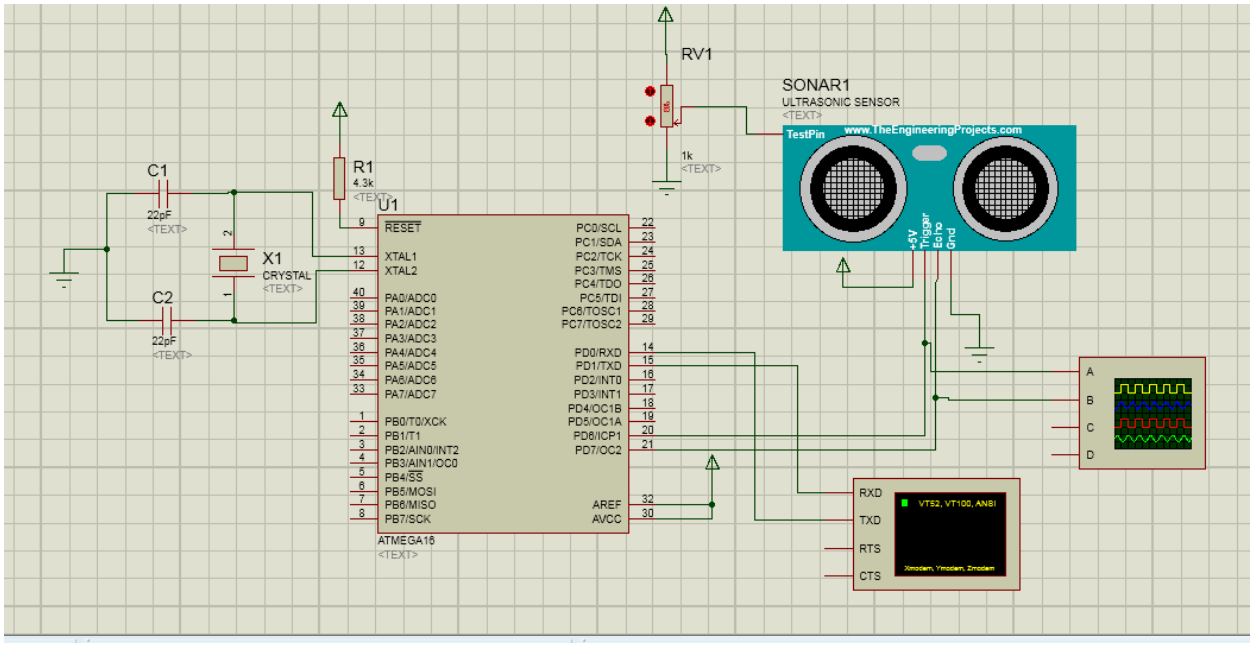


Рис. 5.

Текст програми:

```
#define F_CPU 8000000

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

#define INPUT 0
#define OUTPUT 1

#define LOW 0
#define HIGH 1

#define TRIG_PIN 6
#define ECHO_PIN 7


class PortD
{
public:
PortD();
void setPinMode(char pinNum, char mode);
void setPinState(char pinNum, char level);
char getPinState(char pinNum);
};

PortD::PortD()
{
// There is nothing to initialize here now
}

void PortD::setPinMode(char pinNum, char mode)
{
if(pinNum > 7)
{
	return;
}

if(mode == INPUT)
```

```
{
	DDRD&=~(1<<pinNum);
}
else if(mode == OUTPUT)
{
	DDRD|=(1<<pinNum);
}
}

void PortD::setPinState(char pinNum, char level)
{
if(pinNum > 7)
{
	return;
}

if(level == LOW)
{
	PORTD&=~(1<<pinNum);
}
else if(level == HIGH)
{
	PORTD|=(1<<pinNum);
}
}

char PortD::getPinState(char pinNum)
{
if(pinNum > 7)
{
	return 0;
}

if((PIND&(1<<pinNum)) != 0)
{
	return HIGH;
}

return LOW;
}
```

Клас для роботи з ультразвуковим датчиком выдстаны мыстить вказывник на об’єкт для роботи з портов вводу/виводу, до якого підключений сам датчик, а також два поля для збереження номерів виводів, які використані при підключенні датчика. Слід зазначити, що дана реалізація дозволяє використовувати для підключення датчика бідь-які два виводи в межах одного порта. Також у класі реалізовано конструктор та методи для вимірювання відстанні до перешкоди.

```
class UltrasonicSensor
{
private:
PortD * m_pPortD;
char m_trigPin;
char m_echoPin;
void startMeasure();
public:
UltrasonicSensor(PortD * pPortD, char trigPin, char echoPin);
unsigned int getDistance();
};
```

Реалізація перевантаженого конструктора класу представлена далі. В конструктуор передається вказівник на об’єкт для роботи з портов вводу/виводу (фактично, драйвер порта), та два значення – номери виводів, до яких підключено датчик. Ці аргументи зберігаються у відповідних полях класу.

```
UltrasonicSensor::UltrasonicSensor(PortD * pPortD, char trigPin, char echoPin)
{
m_pPortD = pPortD;
m_trigPin = trigPin;
m_echoPin = echoPin;

m_pPortD->setPinMode(echoPin, INPUT);
m_pPortD->setPinMode(trigPin, OUTPUT);
}
```

Процес вимірювання відстані до перешкоди починається з формування короткого сигналу на виводі Trig. Далі, запускається таймер-лічильник для вимірювання часу, коли датчик прийме відлуння від надісланой пачки імпульсів. Після цього значення, отримане з таймера, перетворюється на фізичні одиниці відстані – в сантиметри.

```
void UltrasonicSensor::startMeasure()
{
char i;

m_pPortD->setPinState(m_trigPin, HIGH);
for(i=0; i<80; i++) __asm("NOP");
m_pPortD->setPinState(m_trigPin, LOW);

// Some pause is needed here!
// During this time the sensor sends 8 pulses on 40kHz frequency.
// For this 200 microseconds is needed and only after that it is necessary to measure the
// pulse width on ECHO pin
for(i=0; i<150; i++) __asm("NOP");

//Start time measure
TCNT1 = 0;
TCCR1B = 0b00000011;
}

unsigned int UltrasonicSensor::getDistance()
{
int timerValue = 0;
int distance;

startMeasure();

while(m_pPortD->getPinState(m_echoPin) == HIGH);

TCCR1B=0;
timerValue = TCNT1;

distance = (int)((double)34 * (double)timerValue/125) / 2; // 125 because the divider for
// timer is 64; if divider is 8 then the value 1000 instead of 125 must be used
return distance;
}

class Uart
{
```

```
public:
Uart();
char receiveChar();
void sendChar(char c);
};

Uart::Uart()
{
UCSRB = 0b00011000;
UCSRC = 0b00000110;
UBRRH = 0;
UBRRL = 51;
}

char Uart::receiveChar()
{
char c;
while((UCSRA & (1<<RXC)) == 0);
c = UDR;
return c;
}

void Uart::sendChar(char c)
{
UDR = c;
while((UCSRA&(1<<UDRE)) == 0);
}
```

В головній функції програми виконується створення об’єкту для роботи з ультразвуковим датчиком відстані, та шляхом виклику його метоів відбувається вимірювання відстані. Отримане значення перетвоюється у ткстовий рядок та відправляється по універсальному асинхронному приймачу/передавачу, дряйвер для роботи з яким було розроблено раніше.

```
int main(void)
{
int distance;
    PortD portD;
UltrasonicSensor sensor(&portD, TRIG_PIN, ECHO_PIN);
Uart uart;

    while (1)
    {
       distance = sensor.getDistance();

       uart.sendChar(distance/1000 + '0');
       uart.sendChar((distance%1000)/100 + '0');
       uart.sendChar(((distance%1000)%100)/10 + '0');
       uart.sendChar(distance%10 + '0');
       uart.sendChar(' ');
       uart.sendChar('c');
       uart.sendChar('m');
       uart.sendChar('\r');

       _delay_ms(500);
    }
}
```

Результати моделювання роботи системи (рис. 6).

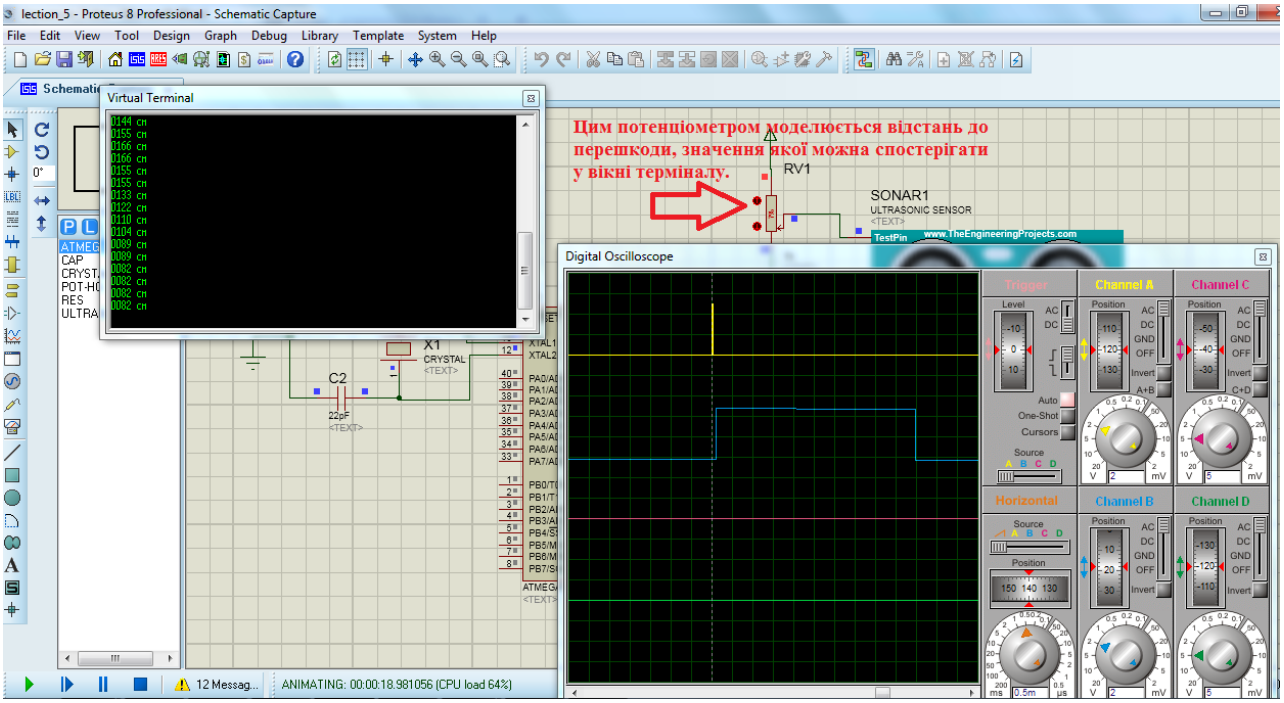


Рис. 6.

Запустивши процес моделювання, у вікні віртуального терміналу можна бачити як змінюється значення виміряної відстані до перешкоди.

Завдання для самостійного виконання:

1. Підключити до мікроконтролера алфавітно-цифровий рідкокристалічний індикатор та доповнити програму класом (тобто розробти клас) для роботи з рідкокристалічним індикатором, на якому відображати виміряну відстань до перешкоди. Модель ультразвукового датчика відстані можна завантажити з сайту за посиланням:

https://github.com/projectiot123/Ultrasonic-Sensor-Library-For-Proteus

Опис принципу роботи датчика можна знайти за посиланням:

https://howtomechatronics.com/tutorials/arduino/ultrasonic-sensor-hc-sr04/

Також інформацію про те, як працює ультразвуковий датчик відстані та як встановити бібліотеку для роботи з цим датчиком у Proteus можна знайти в підручнику «В. М. Рябенький, О. О. Ушкаренко. Програмовані електронні системи керування, збору та обробки інформації», стор. 17-18 та стор. 88-93, що знаходиться на Google-диску в папці «Книги», так само, як і документація на мікроконтролер AtMega16.

2. Доповнити принципову схему мікропроцесорної системи керування мобільною роботизованою платформою шляхом підключення до мікроконтролера ультразвукового датчика відстані. Схема моделі системи керування роботизованою платформою в Proteus розміщено на Google-диску (файл Proteus_model_Lection_6.pdsprj).

3. Доробити програму керування мобільною роботизованою платформою, що було розглянуто на попередньому занятті, наступним алгоритмом: якщо платформа рухається, але відстань до перешкоди стає меншою за 30 см, то рух роботизованої платформи припиняється, тобто всі двигуни зупиняються.

4. Оформити звіт, в якому навести розроблені програми та схеми пристроїв. Завантажити звіт на гугл-диск.

Практична робота 7.

Тема: Принципи розробки класу для роботи з периферійним вузлом аналого-цифрового перетворювача сигналів в мікроконтролері. Організація зворотних зв’язків з використанням датчиків фізичних величин для керування виконавчими механізмами. Приклад програми – мікропроцесорна система керування положенням ротора серводвигуна в залежності від освітленості (розробка класів (драйверів) серводвигуна SG90, класу (драйвера) для роботи з аналого-цифровим перетворювачем; вивчення особливостей передавання адреси порта мікроконтролера в якості параметра конструктора класу, використання ключового слова volatile, робота з вказівниками).

Сервопривід, або привод, що стежить – це механічний привід з автоматичною корекцією стану через внутрішній негативний зворотний зв'язок, відповідно до параметрів, заданих ззовні. На рис. 1 представлено зовнішній вигляд сервоприводу SG90 та осцилограми сигналів керування кутом повороту ротора сервопривода.

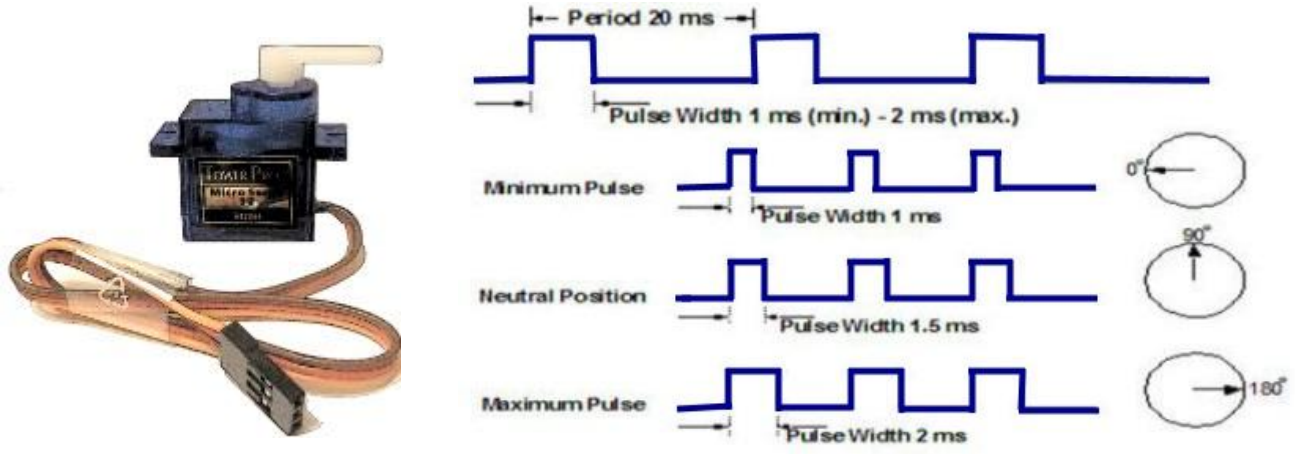


Рис. 1.

Сервопривід SG90 керується ШІМ сигналом, параметри якого визначають положення ротора. Період ШІМ близько 20 мс, тривалість сигналу керування від 1000 до 2000 мкс, хоча в деяких джерелах також вказані значення віз 544 мкс до 2400 мкс. Цей сигнал перетворюється схемою керування, яка вбудована в сервопривод, в напругу. Сервомотор також має вбудований потенціометр, який з'єднаний із вихідним валом. Поворотом валу, сервопривід змінює значення напруги на потенціометрі. Схема керування аналізує напругу вхідного сигналу

(яка пропорційна ширині імпульсу керування), і порівнює її з напругою на потенціометрі. Виходячи з отриманої різниці, мотор плавно обертатиметься до тих пір, поки не вирівняє напругу на виході і на потенціометрі.

Приклад 1. Програма виконує зчитування даних з АЦП (датчика освітеності) та формує сигнал керування на серводвигун SG90 (MG90) для повороту на визначений кут (від 0 до 180 градусів). Призначення системи – керування положенням жалюзі на вікнах для забезпечення постійного рівня освітленості в приміщенні. На рисунках зображено зовнішній вигляд серводвигуна та осцилограми сигналу керування для встановлення заданого кута повороту ротора.

Принципову схему системи керування представлено на рис. 2.

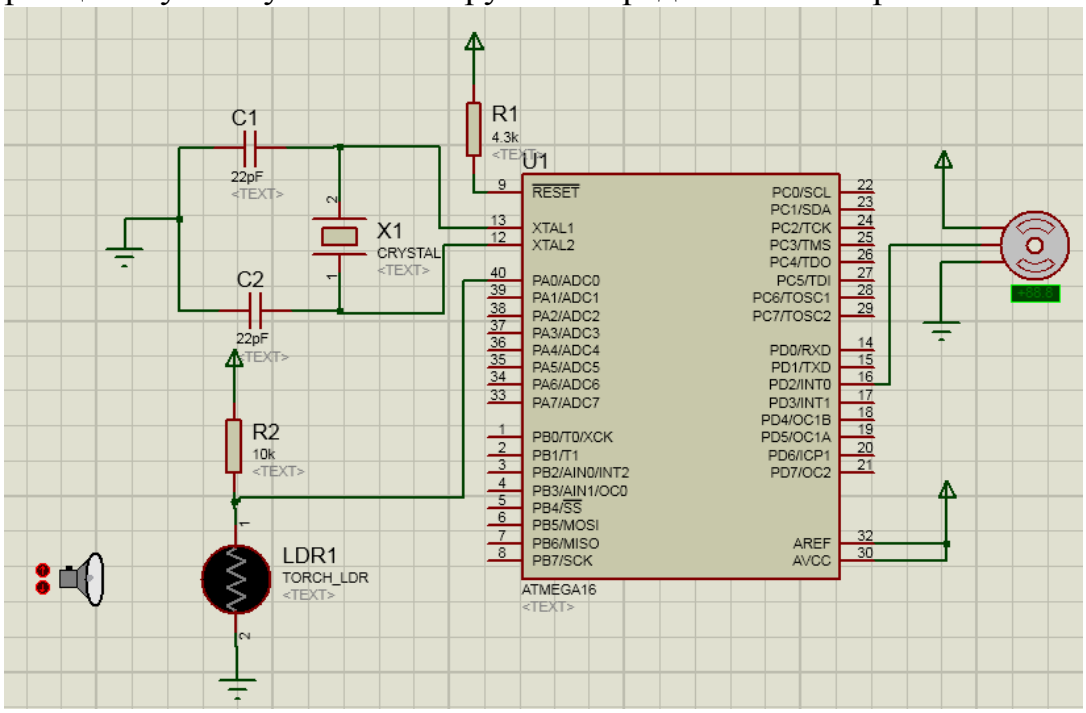


Рис. 2.

Код програми наведено нижче.

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

class Servo
{
private:
volatile unsigned char * m_pPort;
char m_pin;

int convert_angle_to_us(unsigned char angle);
void delay_us(int us);

public:
Servo(volatile unsigned char * port, char pin);
void setAngle(unsigned char angle);
};

Servo::Servo(volatile unsigned char *port, char pin)
{
```

```
m_pPort = port;
m_pin = pin;

*(m_pPort-1) |= (1 << m_pin); // DDRC |= 0b00001000;
}

void Servo::delay_us(int us)
{
for(int i=0; i<us; i++)
{
	__asm("NOP");
}
}

int Servo::convert_angle_to_us(unsigned char angle)
{
int result;

result = 1000 + (int)1000 * ((double)angle / 180);

return result;
}

void Servo::setAngle(unsigned char angle)
{
int us;

if(angle > 180)
{
	angle = 180;
}

us = convert_angle_to_us(angle);

*m_pPort |= (1<<m_pin);
delay_us(us);
*m_pPort &= ~(1<<m_pin);
delay_us(20000 - us);
}


class MyADC
{
public:
MyADC();
int getValue(char channel);
unsigned char getAngle(char channel);
};

MyADC::MyADC()
{
 ADCSRA = 0b10000111;
}

int MyADC::getValue(char channel)
{
char dh, dl;
int result;
	ADMUX = channel;
ADCSRA |= (1 << ADSC); // ADCSRA |= 0b01000000;
```

```
while((ADCSRA & (1 << ADIF)) == 0);
dl = ADCL;
dh = ADCH;

result = dh;
result = result << 8;
result = result + dl;
return result;
}

unsigned char MyADC::getAngle(char channel)
{
int result;
int value;

value = getValue(channel);

result = (int)value * ((double)180 / 1023);
return result;
}

int main(void)
{
      Servo myServo(&PORTD, 2);
MyADC brigthess;

    while (1)
    {
myServo.setAngle(brigthess.getAngle(0));
    }
}
```

Схему мікропроцесорної системи керування представлено на рисунку (файл Proteus_model_Lection_7a.pdsprj на Google-диску).

Приклад 2. Додати в схему, що розглянута в рикладі 1, чотирьохрозрядний семисегментний індикатор та відображати на ньому значення кута повороту ротора сервопривода. Для відображенн даних на індикаторі використовувати динамічну індикацію з частотою оновлення зображення 50 Гц (для цього використовувати таймер/лічильник та переривання).

Принципова схема системи представлена на рис. 3.

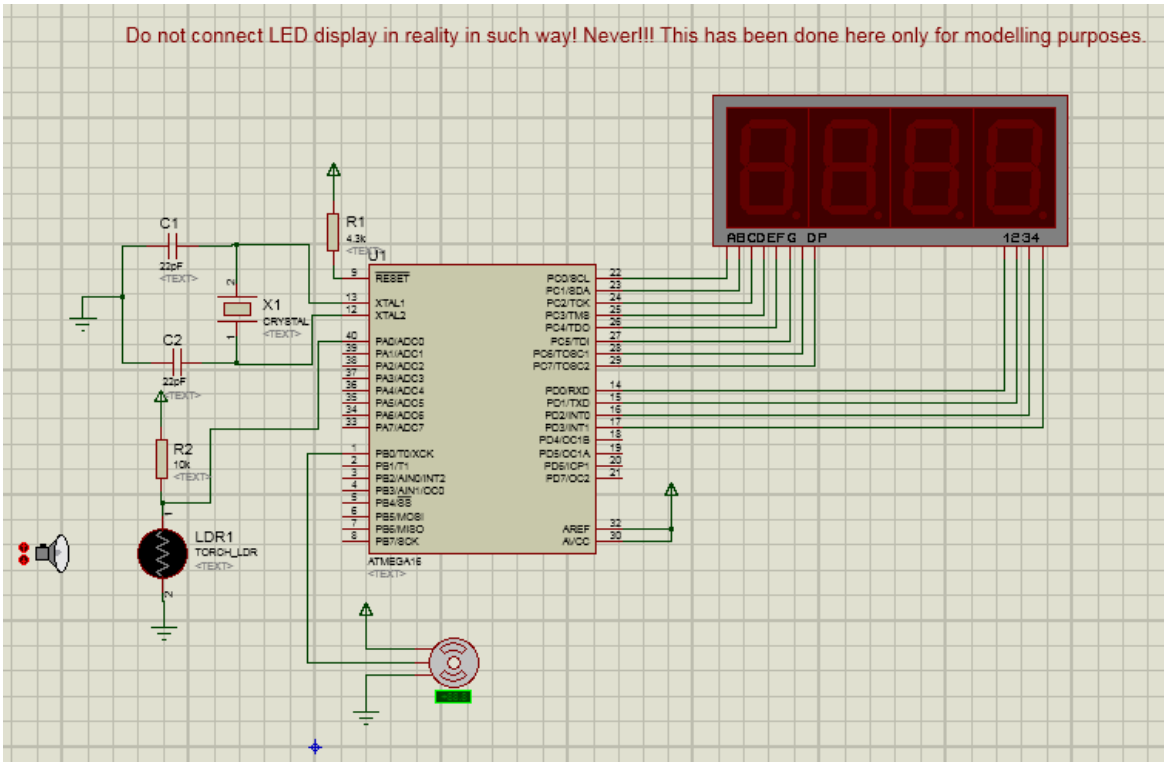


Рис. 3.

Важливо!!! В схемі на рис. 3 виводи 1-4 індикатора підключені напряму до виводів порта. В реальній системі так робити не можна, адже це призведе до виходу мікроконтролера з ладу, тобто його пошкодження, оскільки максимальний струм, що протікає по кожному з виводів буде складати 80 мА (якщо всі сегменти увімкнені). Однак максимальний струм, що може споживатися кожним виводом мікроконтролера, не повинен перевищувати 20 мА. В цій схемі це зроблено лише для моделювання, оскільки при використанні динамічної індикації наявність транзисторів в схемі (або мікросхеми-драйвера, що представляє мобою масив транзисторів Дарлінгтона) на індикаторі нічого відображатися не буде через обмеження системи моделювання Proteus.

Схему мікропроцесорної системи керування представлено на рисунку (файл Proteus_model_Lection_7b.pdsprj на Google-диску).

Текст програми представлено далі.

```
#define F_CPU 8000000
#define MINUS_POS 10

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

В програмі реалізовано динамічне керування семисегментним індикатором. Тому для цього вводиться додаткова структура даних, що зозволить використовувати її для передавання інформації між основною програмою та обробником переривання, в якому виконується оновлення даних.

```
struct DIGITS_TO_SHOW
{
char digit_1;
char digit_2;
char digit_3;
char digit_4;
};
```

Без використання глобальної змінної не вдасться передати інформації з класу в функцію-обробник переривання, що не наледить класу.

```
DIGITS_TO_SHOW digits;
```

Реалізація функції-обробника переривання, в якій виконується виведення даних для відображення на семисегментні індикатори, має вигляд:

```
ISR(TIMER1_OVF_vect)
{
int pos;
pos = PORTD;
switch(pos)
{
	case 0b00001110:
	PORTD = 0b00001101;
	PORTC = digits.digit_2;
	break;

	case 0b00001101:
	PORTD = 0b00001011;
	PORTC = digits.digit_3;
	break;

	case 0b00001011:
	PORTD = 0b00000111;
	PORTC = digits.digit_4;
	break;

	case 0b000000111:
	PORTD = 0b00001110;
	PORTC = digits.digit_1;
	break;

	default:
	PORTD = 0b00001110;
	break;
}

TCNT1 = 64910;
}
```

Клас-драйвер семисегментного індикатора наведено далі. В класі реалізовано конструктор за замовчуванням та функцію, що виконує виведення даних на індикатор.

```
class Indicator
{
public:
```

```
Indicator();
void UpdateValue(int value);
};
```

В конструкторі за замовчуванням виконується налаштування портів введення/виведення у відповідності до принципової схеми, а також запус таймера/лічильника, що використовується для реалізації динамічної ідникації. Оновлення інформації має відбуватися з частотою не менше 50 Гц щоб око не бачило мерехтіння. З урахуванням кількості розрядів індикатора (4 знакомісця), період роботи таймера має складати 1/200 (с) = 5 (мс).

```
Indicator::Indicator()
{
DDRC = 0xFF;
DDRD = 0x0F;

TCNT1 = 64910;
TIMSK = (1 << TOIE1);
TCCR1B = 0b00000011;

sei();
}
```

В функції, що виконує виведення даних на індикатор, програмно реалізовано дешифратор двійкового коду в код семисегментного індикатора.

```
void Indicator::UpdateValue(int value)
{
const char m_seg[11] = {0b00111111, 0b00000110, 0b01011011, 0b01001111, 0b01100110,
0b01101101, 0b01111101, 0b00000111, 0b01111111, 0b01101111, 0b01000000};

if(value >= 0)
{
	digits.digit_1 = m_seg[value/1000];
}
else
{
	digits.digit_1 = m_seg[MINUS_POS];
	value = value * (-1);
}

digits.digit_2 = m_seg[(value%1000)/100];
digits.digit_3 = m_seg[(value%100)/10];
digits.digit_4 = m_seg[value%10];
}

class Servo
{
private:
volatile unsigned char * m_pPort;
char m_pin;

int convert_angle_to_us(unsigned char angle);
void delay_us(int us);

public:
Servo(volatile unsigned char * port, char pin);
```

```
void setAngle(unsigned char angle);
};

Servo::Servo(volatile unsigned char *port, char pin)
{
m_pPort = port;
m_pin = pin;

*(m_pPort-1) |= (1 << m_pin); // DDRC |= 0b00001000;
}

void Servo::delay_us(int us)
{
for(int i=0; i<us; i++)
{
	__asm("NOP");
}
}

int Servo::convert_angle_to_us(unsigned char angle)
{
int result;

result = 1000 + (int)1000 * ((double)angle / 180);

return result;
}

void Servo::setAngle(unsigned char angle)
{
int us;

if(angle > 180)
{
	angle = 180;
}

us = convert_angle_to_us(angle);

*m_pPort |= (1<<m_pin);
delay_us(us);
*m_pPort &= ~(1<<m_pin);
delay_us(20000 - us);
}

class MyADC
{
public:
MyADC();
int getValue(char channel);
unsigned char getAngle(char channel);
};

MyADC::MyADC()
{
ADCSRA = 0b10000111;
}

int MyADC::getValue(char channel)
{
```

```
char dh, dl;
int result;

ADMUX = channel;
ADCSRA |= (1 << ADSC); // ADCSRA |= 0b01000000;
while((ADCSRA & (1 << ADIF)) == 0); // while((ADCSRA  &0b00010000) == 0)
dl = ADCL;
dh = ADCH;

result = dh;
result = result << 8;
result = result + dl;
return result;
}

unsigned char MyADC::getAngle(char channel)
{
int result;
int value;

value = getValue(channel);

result = (int)value * ((double)180 / 1023);
return result;
}
```

Головна функція програми, в якій використовуються розроблені класи.

```
int main(void)
{

Servo myServo(&PORTB, 0);
MyADC brigthess;
Indicator indicator;
int angle;

while (1)
{
	angle = brigthess.getAngle(0);
	myServo.setAngle(angle);
	indicator.UpdateValue(angle - 90);
	_delay_ms(200);
}
}
```

В режимі моделювання можна змінити положення джерела світла і побачити, що ротор серводвигуна обертається, а на семисегментном індикаторі відображатиметься його поточне значення кута обертання.

Завдання для самостійного виконання:

1. Підключити до мікропроцесорної системи керування, що наведена в прикладі, алфавітно-цифровий рідкокристалічний індикатор та відображати на ньому значення кута повороту ротора сервопривода. Можна використовувати клас для роботи з РКІ, що розглянутий на попередніх заняттях. Схема моделі системи керування, що розглянута в прикладі, розміщено на Google-диску (файл Proteus_model_Lection_7.pdsprj). Допускається похибка у відображенні значення кута на РКІ та фактичного значення кута повороту ротора сервопривода до 5%.

2. Убрати зі схеми датчик освітленості і підключити до мікроконтролера дві кнопки. Якщо жодна з кнопок не натиснута, то кут повороту ротора серводвигуна має складати 90 градусів. Якщо натиснута перша кнопка, то кут повороту ротора серводвигуна має бути в межах 0-10 градусів. Якщо натиснута друга кнопка, то кут повороту ротора серводвигуна повинен бути в межах 170-180 градусів.

3. Розглянути (за бажанням повторити) приклад керування серводвигуном для Ардуіно, який можна знайти в підручнику «В. М. Рябенький, О. О. Ушкаренко. Програмовані електронні системи керування, збору та обробки інформації», стор. 53-55, що знаходиться на Google-диску в папці «Книги».

4. Оформити звіт, в якому навести розроблені програми та схеми пристроїв. Завантажити звіт на гугл-диск.

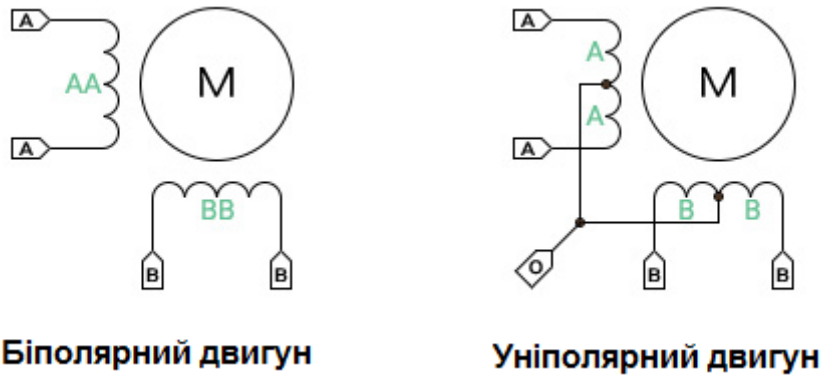
Практична робота 8.

Тема: Розробка класів (драйверів) для керування уніполярним та біполярним крокovими двигунами; вивчення особливостей передавання адреси порта мікроконтролера в якості параметра конструктора класу, використання ключового слова volatile, робота з вказівниками.

Крокovий двигун – це безколекторний двигун, ротор якого обертається не плавно, а кроками (дискретно). Повний оборот ротора складається з кількох кроків. Змінюючи форму сигналу, кількість імпульсів, їх тривалість і фазовий зсув, можна задавати швидкість обертання, напрямок обертання та кількість обертів ротора двигуна.

Крокові двигуни складаються з ротора (рухлива частина) та статора (нерухома частина). На статорі встановлюють електромагніти, а частини ротора, що взаємодіють з електромагнітами, виконуються з магнітотвердого (двигун з постійними магнітами) або магнітом'якого (реактивний двигун) матеріалу.

За типом ротора, крокові двигуни поділяються на: двигуни з постійними магнітами, реактивні двигуни та гібридні двигуни. За типом з'єднання електромагнітів, крокові двигуни поділяються на: уніполярні та біполярні (рис. 1).



Біполярний двигун має 4 виводи. Виводи А живлять обмотку АА, виводи В живлять обмотку ВВ. Для включення електромагніту на виводи обмотки необхідно подати різницю потенціалів (два різних рівня), тому двигун називається

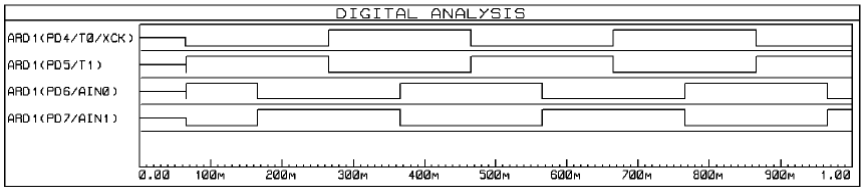
біполярним. Напрямок магнітного поля залежить від полярності потенціалів на виводах.

Уніполярний двигун має 5 (іноді 6) виводів. Центральні точки його обмоток з'єднані між собою і є загальним (п'ятим) виводом, який зазвичай (але не завжди) підключають до GND. Для включення електромагніту достатньо подати позитивний потенціал на один з виводів обмотки, тому двигун називається уніполярним. Напрямок магнітного поля залежить від того, на який саме вивід обмотки подано позитивний потенціал.

Приклад 1. Програма виконує керування процесом обертання ротора уніполярного крокового двигуна. На рис. 2, а, б, зображено зовнішній вигляд крокового двигуна та сигнали керування, які необхідно подавати на обомотки крокового двигуна.

Уніполярний кроковий двигун з платою керування ULN2003

а



Осцилограма сигналів керування

б

Рис. 2.

Код програми наведено нижче.

```
#define F_CPU 8000000

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

class UnipolarStepper
{
private:

enum STATE{STATE_1, STATE_2, STATE_3, STATE_4};

volatile unsigned char *m_pPort;
char m_pin1;
char m_pin2;
char m_pin3;
char m_pin4;

STATE m_state;

void setPinStates();

public:
UnipolarStepper(volatile unsigned char *port, char pin1, char pin2, char pin3, char pin4);
void StepCW(); // clockwise - за годинниковою стрілкою
```

```
void StepCCW(); // counter clockwise - проти годинникової стрілки
void Rotate(int angle, double delta);
};

UnipolarStepper::UnipolarStepper(volatile unsigned char *port, char pin1, char pin2, char
pin3, char pin4)
{
m_pPort = port;
m_pin1 = pin1;
m_pin2 = pin2;
m_pin3 = pin3;
m_pin4 = pin4;
m_state = STATE_1;

*(m_pPort-1) |= ((1<<m_pin1) | (1<<m_pin2) | (1<<m_pin3) | (1<<m_pin4));
// DDRD = 0b11110000;
}

void UnipolarStepper::StepCW()
{
switch(m_state)
{
	case STATE_1:
		m_state = STATE_2;
	break;
	case STATE_2:
		m_state = STATE_3;
	break;
	case STATE_3:
		m_state = STATE_4;
	break;
	case STATE_4:
		m_state = STATE_1;
	break;
}
setPinStates();
}

void UnipolarStepper::StepCCW()
{
switch(m_state)
{
	case STATE_1:
	m_state = STATE_4;
	break;
	case STATE_2:
	m_state = STATE_1;
	break;
	case STATE_3:
	m_state = STATE_2;
	break;
	case STATE_4:
	m_state = STATE_3;
	break;
}
setPinStates();
}

void UnipolarStepper::Rotate(int angle, double delta)
{
unsigned int steps = (unsigned int)abs(angle) / delta;
for(unsigned int i=0;i<steps;i++)
{
```

```
        if(angle < 0)
        {
                StepCCW();
        }
        else
        {
                StepCW();
        }
        _delay_ms(250);
}
}


void UnipolarStepper::setPinStates()
{
switch(m_state)
{
        case STATE_1:
                *m_pPort = (1<<m_pin2) | (1<<m_pin3);
        break;
        case STATE_2:
                *m_pPort = (1<<m_pin2) | (1<<m_pin4);
        break;
        case STATE_3:
                *m_pPort = (1<<m_pin1) | (1<<m_pin4);
        break;
        case STATE_4:
                *m_pPort = (1<<m_pin1) | (1<<m_pin3);
        break;
}
}

int main(void)
{
    UnipolarStepper myStepper(&PORTD, 4, 5, 6, 7);

    //myStepper.Rotate(-42, 2);

    while (1)
    {
       myStepper.StepCCW();
       _delay_ms(250);
    }
}
```

Схему мікропроцесорної системи керування представлено на рисунку (файл Proteus_model_Lection_8.pdsprj на Google-диску). На схемі зверху праворуч розташовано біполярний крововий двигун, а знизу праворуч – уніполярний крововий двигун (саме для цього двигуна наведено код програми в цьому прикладі).

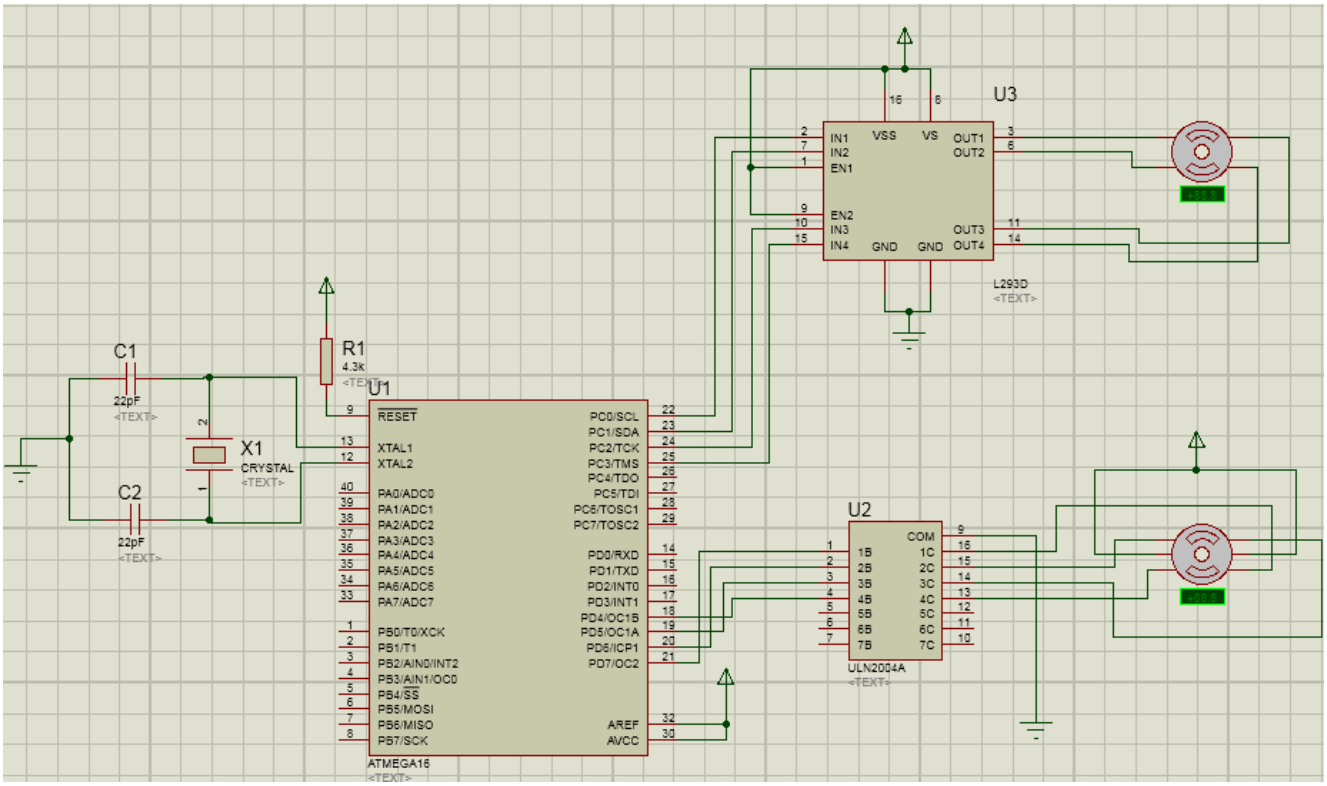


Рис. 3.

Електромотори не можна підключати до висновків Arduino безпосередньо, оскільки вони споживають значні струми, крокові двигуни є винятком, тому їх підключають через драйвери. Біполярний двигун можна підключити лише до драйвера біполярних двигунів. 6-вивідний двигун можна підключити до будь-якого драйвера. Якщо не використовувати виводи центральних точок обмоток, двигун буде біполярним, а якщо ці виводи з'єднати і підключити до GND (або VCC, в залежності від мікросхеми-драйвера), то двигун буде уніполярним.

Приклад 2. Програма виконує керування процесом обертання ротора біполярного крокового двигуна (на схемі, що наведена вище, двигун розташований зверху праворуч). На рис. 4 наведено осцилограму сигналів керування.

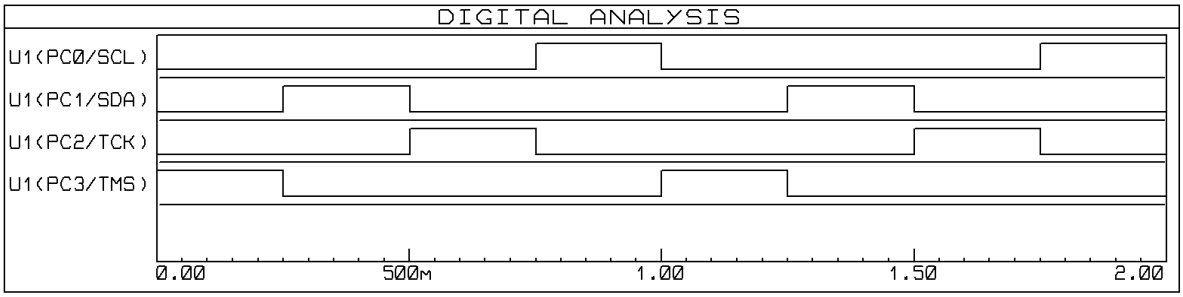


Рис. 4.

Текст програми представлено далі.

```
#define F_CPU 8000000

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

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


class BipolarStepper
{
private:

enum STATE{STATE_1, STATE_2, STATE_3, STATE_4};

volatile unsigned char *m_pPort;
char m_pin1;
char m_pin2;
char m_pin3;
char m_pin4;

STATE m_state;

void setPinStates();

public:
BipolarStepper(volatile unsigned char *port, char pin1, char pin2, char pin3, char pin4);
void StepCW(); // clockwise - за годинниковою стрілкою
void StepCCW(); // counter clockwise - проти годинникової стрілки
void Rotate(int angle, double delta);
};

BipolarStepper::BipolarStepper(volatile unsigned char *port, char pin1, char pin2, char pin3,
char pin4)
{
m_pPort = port;
m_pin1 = pin1;
m_pin2 = pin2;
m_pin3 = pin3;
m_pin4 = pin4;
m_state = STATE_1;

*(m_pPort-1) |= ((1<<m_pin1) | (1<<m_pin2) | (1<<m_pin3) | (1<<m_pin4));
}

void BipolarStepper::StepCW()
{
switch(m_state)
{
	case STATE_1:
	m_state = STATE_2;
	break;
	case STATE_2:
	m_state = STATE_3;
	break;
	case STATE_3:
	m_state = STATE_4;
	break;
	case STATE_4:
	m_state = STATE_1;
	break;
}
setPinStates();
}


void BipolarStepper::StepCCW()
{
```

```
switch(m_state)
{
	case STATE_1:
	m_state = STATE_4;
	break;
	case STATE_2:
	m_state = STATE_1;
	break;
	case STATE_3:
	m_state = STATE_2;
	break;
	case STATE_4:
	m_state = STATE_3;
	break;
}
setPinStates();
}

void BipolarStepper::Rotate(int angle, double delta)
{
unsigned int steps = (unsigned int)abs(angle) / delta;
for(unsigned int i=0;i<steps;i++)
{
	if(angle < 0)
	{
		StepCCW();
	}
	else
	{
		StepCW();
	}
	_delay_ms(250);
}
}

void BipolarStepper::setPinStates()
{
switch(m_state)
{
	case STATE_1:
		*m_pPort = (1<<m_pin1);
	break;
	case STATE_2:
		*m_pPort = (1<<m_pin3);
	break;
	case STATE_3:
		*m_pPort = (1<<m_pin2);
	break;
	case STATE_4:
		*m_pPort = (1<<m_pin4);
	break;
}
}


int main(void)
{
	BipolarStepper biStepper(&PORTC, 0, 1, 2, 3);

//biStepper.Rotate(-48, 2);

    while (1)
    {
```

```
        biStepper.StepCCW();
        _delay_ms(250);
    }
}
```

Завдання для самостійного виконання:

1. Підключити до мікропроцесорної системи керування, що наведена в прикладі, віртуальний трмінал для того, щоб можна було б передавати команди керування кроковим двигуном. Схема моделі системи керування, що розглянута в прикладі, розміщено на Google-диску (файл Proteus_model_Lection_8.pdsprj).

2. Розробити клас для роботи з універсальним асинхронним приймачем/передавачем (USART). Параметри обміну даними наступні: швидкість 9600 біт/с, 8 біт даних, 1 стоповий біт, перевірка на парність відсутня. Реалізувати функцію для зситування отриманого байту (символу) по USART.

3. Інтегрувати розроблений клас для роботи з універсальним асинхронним приймачем/передавачем (USART) в систему керування кроковим двигуном. При надходженні по USART символу ‘S’ або ‘s’ кроковий двигун повинен зробити один крок за годинниковою стрілкою (тобто положення ротора двигуна збільшиться на 2 градуси). При отриманні символу ‘C’ або ‘c’ кроковий двигун повинен зробити один крок проти годинникової стрілки (тобто положення ротора двигуна зменшиться на 2 градуси).

4. Розглянути (за бажанням повторити) приклад керування кроковим двигуном для Ардуіно, який можна знайти в підручнику «В. М. Рябенький, О. О. Ушкаренко. Програмовані електронні системи керування, збору та обробки інформації», стор. 55-59, що знаходиться на Google-диску в папці «Книги».

4. Оформити звіт, в якому навести розроблені програми та схеми пристроїв. Завантажити звіт на гугл-диск.

Практична робота 9.

Тема: Принципи використання успадковування класів на прикладі розробки мікропроцесорної системи керування уніполярним та біполярним кроковими двигунами (наслідування, перевантаження функцій, перевизначення функцій-членів класів).

Успадкуовування (inheritance) є одним з ключових аспектів об'єктно-орієнтованого програмування, який дозволяє успадковувати функціональність одного класу (базового класу) в іншому – похідному класі (derived class).

Для встановлення відношення успадкування після назви класу ставиться двокрапка, потім йде специфікатор доступу та назва класу, від якого ми хочемо успадкувати функціональність. Щодо цього клас AbstractStepper в наступному прикладі ще називатиметься базовим класом (також називають суперкласом, батьківським класом), а UnipolarStepper та BipolarStepper – похідними класом (також називають підкласами, класами-спадкоємцями).

Специфікатор доступу дозволяє вказати, до яких членів класу похідний клас матиме доступ. Якщо використовується специфікатор public, то він дозволяє використовувати у похідному класі всі відкриті члени базового класу. Якщо ми не

використовуємо модифікатор доступу, то класи UnipolarStepper та BipolarStepper нічого не знатимуть про змінні m_state, m_pPort, m_pin1, m_pin2, m_pin3, m_pin4, які є полями базового класу AbstractStepper. Після встановлення спадкування ми можемо прибрати з класів UnipolarStepper та BipolarStepper ті змінні, які вже визначені у класі AbstractStepper.

Приклад. В програмі реалізовано один базовий клас та два похідні класи. Похідні класи реалізують керування процесом обертання роторів відповідно уніполярного та біполярного крокових двигунів. На рис. 1 представлена ієрархія класів.

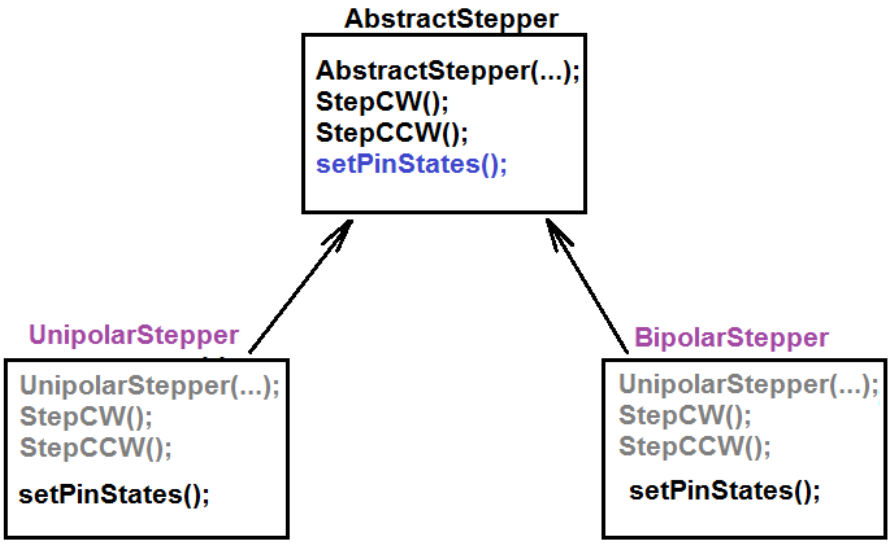


Рис. 1.

Розробка починається зі створення абстрактного класу, в якому міститься спільна інформація, що не залежить від типу крокового двигуна. Цей клас реалізує інтерфейс керування двигунами, а також містить поля для зберігання адреси порта та номери виводів, до якого підключений двигун. В конструкторі цього класу відбувається ініціалізація полів та налаштування відповідних виводів порта на вихід. Методи класа надають можливість керування двигунов – зробити один крок за годинниковою стрілкою, один крок проти годинникової стрілки та повернути ротор на заданий кут. Код програми наведено нижче.

```
#define F_CPU 8000000

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

class AbstractStepper
{
protected:

enum STATE{STATE_1, STATE_2, STATE_3, STATE_4};

volatile unsigned char *m_pPort;
STATE m_state;
char m_pin1;
char m_pin2;
char m_pin3;
```

```
char m_pin4;

public:
AbstractStepper(volatile unsigned char * port, char pin1, char pin2, char pin3, char pin4);
void StepCW(); // clock wise - за годинниковою стрілкою
void StepCCW();
void Rotate(int angle, char delta);
void setPinStates(); // pure virtual function
};
```

В конструкторі класу відбувається ініціалізація полів класу та налаштування виводів порта вводу/виводу, до яких підключено двигун, на вихід.

```
AbstractStepper::AbstractStepper(volatile unsigned char * port, char pin1, char pin2, char
pin3, char pin4)
{
m_pPort = port;
m_pin1 = pin1;
m_pin2 = pin2;
m_pin3 = pin3;
m_pin4 = pin4;
m_state = STATE_1;

*(m_pPort - 1) |= ((1<<m_pin1) | (1<<m_pin2) | (1<<m_pin3) | (1 << m_pin4)); // 0b00001111
}
```

В наступній функції відбувається формування сигналів керування, що призводять до повороту ротора крокового двигуна на один крок за годинниковою стрілкою.

```
void AbstractStepper::StepCW()
{
switch(m_state)
{
	case STATE_1:
		m_state = STATE_2;
	break;
	case STATE_2:
		m_state = STATE_3;
	break;
	case STATE_3:
		m_state = STATE_4;
	break;
	case STATE_4:
		m_state = STATE_1;
	break;
	default:
		m_state = STATE_1;
	break;
}
}
```

В наступній функції відбувається формування сигналів керування, що призводять до повороту ротора крокового двигуна на один крок проти годинникової стрілки.

```
void AbstractStepper::StepCCW()
{
switch(m_state)
{
```

```
        case STATE_1:
        m_state = STATE_4;
        break;
        case STATE_2:
        m_state = STATE_1;
        break;
        case STATE_3:
        m_state = STATE_2;
        break;
        case STATE_4:
        m_state = STATE_3;
        break;
        default:
        m_state = STATE_1;
        break;
}
}
```

Принцип роботи функції, в якій формуються сигнали керування для повороту ротора крокового двигуна полягає у визначенні кількості кроків, а потім у циклі викликається функція формування сигналів для здійснення одного кроку. В залежності від знаку кута повороту, в циклі відбувається виклик функції, що робить один крок за годинниковою стрілкою або проти годинникової стрілки.

```
void AbstractStepper::Rotate(int angle, char delta)
{
unsigned int steps = (unsigned int)abs(angle) / delta;
for(unsigned int i = 0; i<steps; i++)
{
        if(angle < 0)
        {
                StepCCW();
        }
        else
        {
                StepCW();
        }
        _delay_ms(250);
}
}
```

Оскільки сигнали керування уніполярним та біполярним двигунами різні, то в самій функції, що змінює стани сигналів на виводі порта, та яка належить абстрактному класу, немає потреби виконувати якісь дії. Це буде реалізовано в похідних класах.

```
void AbstractStepper::setPinStates()
{
// Dummy - заглушка
}
```

Клас для керування біполярним кроковим двигуном наслідується від абстрактного класу. Варто враховувати, що конструктори при наслідуванні не успадковуються. І якщо базовий клас містить лише конструктори з параметрами, то похідний клас, як в даному прикладі, повинен викликати у своєму конструкторі один із конструкторів базового класу. Після списку параметрів конструктора похідного класу через двокрапку йде виклик конструктора базового класу, який

передаються значення параметрів. Спочатку буде викликатись конструктор базового класу AbstractStepper, у який будуть передаватися значення port, pin1, pin2, pin3, pin4. І таким чином будуть вказані порт мікроконтролера та номери виводів, до яких підключено кроковий двигун. Потім виконуватиметься власне конструктор BipolarStepper.

```
class BipolarStepper : public AbstractStepper
{
public:
BipolarStepper(volatile unsigned char *port, char pin1, char pin2, char pin3, char pin4) :
        AbstractStepper(port, pin1, pin2, pin3, pin4) {
        // Nothing add more
        };
void StepCW();
void StepCCW();
void Rotate(int angle, char delta);
void setPinStates();
};
```

У функціях, що формують сигнали на двигун для обертання на один крок за годинниковою стрілкою або проти годинникової стрілки відбувається виклик відповідної функції абстрактного класу, що призводить до зміни стану цифрового автомату. А потім викликається функція, що безпосередньо змінює сигнали на виводах порта мікроконтролера в залежності від поточного стану цифрового автомату.

```
void BipolarStepper::StepCW()
{
AbstractStepper::StepCW();
setPinStates();
}

void BipolarStepper::StepCCW()
{
AbstractStepper::StepCCW();
setPinStates();
}

void BipolarStepper::Rotate(int angle, char delta)
{
unsigned int steps = (unsigned int)abs(angle) / delta;
for(unsigned int i = 0; i<steps; i++)
{
       if(angle < 0)
       {
              StepCCW();
       }
       else
       {
              StepCW();
       }
       _delay_ms(250);
}
}
```

В наступній функції відбувається керування безпосередньо рівнями сигналів на виводах порта мікроконтролера, до яких підключений біполярний

кроковий двигун, в залежності від поточного страну цифрового автомату. Ця частина програмного коду залежить від типу крокового двигуна, що використовується, тому не могла бути реалізована в абстрактному класі.

```
void BipolarStepper::setPinStates()
{
switch(m_state)
{
	case STATE_1:
		*m_pPort = (1<<m_pin1);
	break;
	case STATE_2:
		*m_pPort = (1<<m_pin3);
	break;
	case STATE_3:
		*m_pPort = (1<<m_pin2);
	break;
	case STATE_4:
		*m_pPort = (1<<m_pin4);
	break;
	default:
		*m_pPort = 0;
	break;
}
}
```

Клас для керування уніполярним кроковим двигуном наслідується від абстрактного класу.

```
class UnipolarStepper : public AbstractStepper
{
public:
UnipolarStepper(volatile unsigned char *port, char pin1, char pin2, char pin3, char pin4) :
	 AbstractStepper(port, pin1, pin2, pin3, pin4) {
	// Nothing add more
};
```

У функціях, що формують сигнали на двигун для обертання на один крок за годинниковою стрілкою або проти годинникової стрілки відбувається виклик відповідної функції абстрактного класу, що призводить до зміни стану цифрового автомату. А потім викликається функція, що безпосередньо змінює сигнали на виводах порта мікроконтролера в залежності від поточного стану цифрового автомату.

```
void StepCW();
void StepCCW();
void Rotate(int angle, char delta);
void setPinStates();
};

void UnipolarStepper::StepCW()
{
AbstractStepper::StepCW();
setPinStates();
}

void UnipolarStepper::StepCCW()
{
AbstractStepper::StepCCW();
```

```
setPinStates();
}

void UnipolarStepper::Rotate(int angle, char delta)
{
unsigned int steps = (unsigned int)abs(angle) / delta;
for(unsigned int i = 0; i<steps; i++)
{
	if(angle < 0)
	{
		StepCCW();
	}
	else
	{
		StepCW();
	}
	_delay_ms(250);
}
}
```

В наступній функції відбувається керування безпосередньо рівнями сигналів на виводах порта мікроконтролера, до яких підключений біполярний крокових двигун, в залежності від поточного страну цифрового автомату. Ця частина програмного коду залежить від типу крокового двигуна, що використовується, тому не могла бути реалізована в абстрактному класі.

```
void UnipolarStepper::setPinStates()
{
switch(m_state)
{
	case STATE_1:
	*m_pPort = (1<<m_pin2) | (1<<m_pin3);
	break;
	case STATE_2:
	*m_pPort = (1<<m_pin2) | (1<m_pin4);
	break;
	case STATE_3:
	*m_pPort = (1<<m_pin4) | (1<<m_pin1);
	break;
	case STATE_4:
	*m_pPort = (1<<m_pin1) | (1<<m_pin3);
	break;
	default:
	*m_pPort = 0;
	break;
}
}
```

В головній функції програми відбувається створення екземплярів класів для роботи з уніполярним та біполярним кроковими двигунами, та виклик функції Rotate для перевірки коректності реалізації програми.

```
int main(void)
{

BipolarStepper biStepper(&PORTC, 0, 1, 2, 3);
biStepper.Rotate(-30, 2);
biStepper.Rotate(60, 2);
biStepper.Rotate(-30, 2);
```

```
UnipolarStepper uniStepper(&PORTD, 4, 5, 6, 7);
uniStepper.Rotate(-30, 2);
uniStepper.Rotate(60, 2);
uniStepper.Rotate(-30, 2);

    while (1)
    {

    }
}
```

Варто враховувати, що конструктори при наслідуванні не успадковуються. І якщо базовий клас містить лише конструктори з параметрами, як в прикладі, то похідний клас повинен викликати у своєму конструкторі один із конструкторів базового класу.

Після списку параметрів конструктора похідного класу через двокрапку йде виклик конструктора базового класу, який передаються значення параметрів.

Принципова схема мікропроцесорної системи керування (файл Proteus_model_Lection_9.pdsprj) представлена на рис. 2. При моделюванні роботи системи можна побачити, що спочатку ротор біполярного крокового двигуна повернеться на 30 градусів проти годинникової стрілки, потім на 60 градусів да годинниковою стрілкою, і потім на 30 градусів проти годинниково стрілки та займе своє початкове положення. Те ж саме відбудеться з ротором уніполярного крокового двигуна, що підтверджує коректність роботи програми.

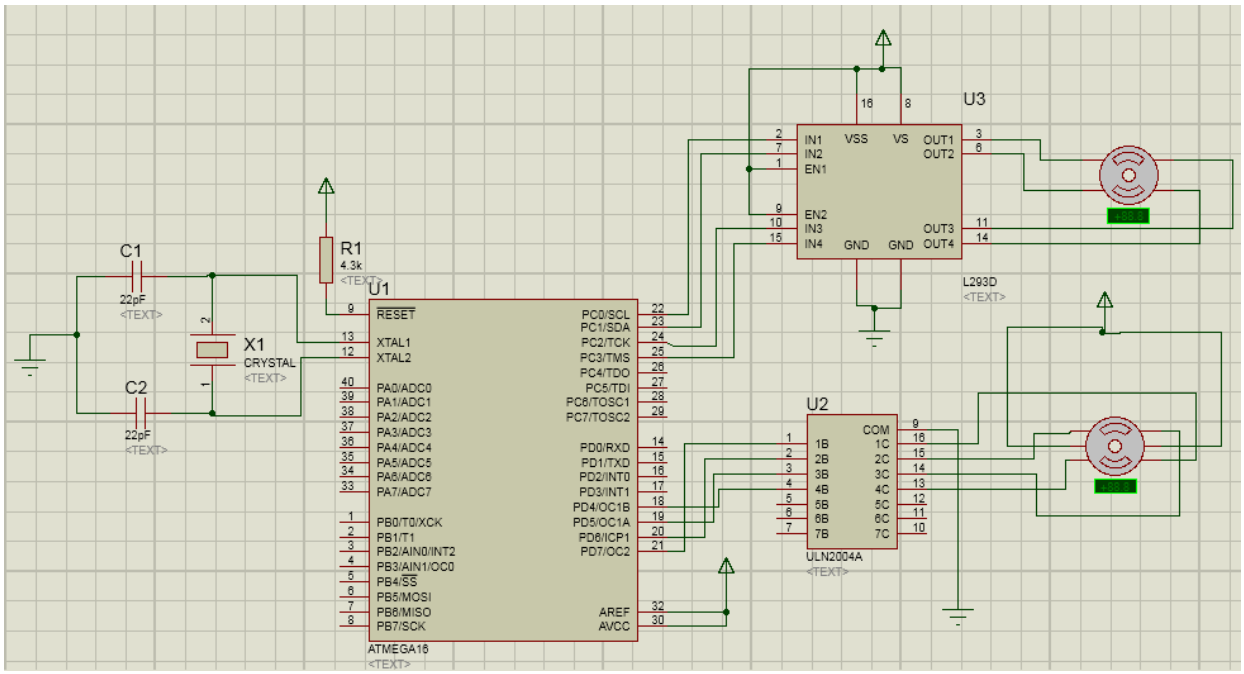


Рис. 2.

Успадкування корисне, оскільки воно дозволяє структурувати та повторно використовувати код, що, у свою чергу, може значно прискорити процес розробки. Незважаючи на це, успадкування слід використовувати з обережністю, оскільки більшість змін у суперкласі торкнуться всіх підкласів, що може призвести до непередбачених наслідків.

C++, на відміну деяких ООП мов, не надає окремого ключового слова позначення інтерфейсу (interface). Тим не менш, реалізація інтерфейсу можлива шляхом створення чистого абстрактного класу (pure abstract class) – класу, в якому присутні лише декларації методів. Такі класи часто називають абстрактними базовими класами (Abstract Base Class — ABC).

У C++ клас, у якому існує хоча б один чистий віртуальний метод (pure virtual), прийнято вважати абстрактним. Якщо віртуальний метод не перевизначено у дочірньому класі, код не скомпілюється. Також, у C++ створити об'єкт абстрактного класу неможливо – спроба також спричинить помилку при компіляції. Однак слід зауважити, що не всі середовища розробки (компілятори) програмного забепечення вбудованих мікропроцесорних систем підтримують механізм віртуальних функцій в C++.

Завдання для самостійного виконання:

1. Розробити принципову схему системи керування світлодіодами. До виводів мікроконтролера підключити 3 світлодіоди – зеленого, червоного та синього кольорів.

2. Розробити ієрархію класів, яка складається з чотирьох класів – базового та трьох похідних класів. Кожен з похідних класів представляє собою драйвер світлодіода відповідного кольору. Реалізувати функції-члени класів для вмикання та вимикання світлодіода. Перевірити роботу програми.

3. Підключити до входів АЦП мікроконтролера аналогові датчики освітленості (Torch) та датчик температури (LM35). Опис датчика температури можна знайти за посиланням:

https://www.hwlibre.com/ru/lm35/

Розробити базовий абстрактний клас AbstractSensor та похідні класи LightSensor та TemperatureSensor. Також потрібно реалізувати клас-драйвер для роботи з АЦП. Реалізувати функції-члени класів для зчитування даних з датчиків та передавати виміряні значення по UART. Для цього потрібно буде також розробити клас-драйвер для роботи з UART.

4. Оформити звіт, в якому навести розроблені програми та схеми пристроїв. Завантажити звіт на гугл-диск.

Практична робота 10.

Тема: Вивчення принципів наслідування класів для запобігання дублюванню програмного коду. Приклад програми – мікропроцесорна система вимірювання концентрації різних газів у приміщенні (розробка класів (драйверів) датчиків газів; розробка драйверів АЦП та універсального асинхронного приймача/передавача).

Датчики присутності газів потрібні для виявлення витоку природних і побутових газів. Датчики реагують на виникнення в повітрі домішок газу. При

появі горючого газу провідність датчика зростає з ростом концентрації цього газу. Використовуючи просту електронну схему, можна перетворити провідність датчика в сигнал, пропорційний концентрації газу. Чутливий матеріал датчиків газу серії MQ – це високоактивний оксид металу, зазвичай SnO_2. Коли чутлива поверхня датчика нагрівається до певної температури, атоми кисню абсорбуються поверхнею напівпровідника, насиченого електронами.

Датчик газу (рис. 1), побудований на базі газоаналізатора MQ-2 дозволяє виявляти наявність в навколишньому повітрі вуглеводневих газів (пропан, метан, бутан), диму (зважені частинки, які є результатом горіння), водню.

Датчик можна використовувати для виявлення витоків промислового газу і задимлення. Вихідним результатом є аналоговий сигнал, пропорційний вмісту газів, до яких чутливий газоаналізатор.

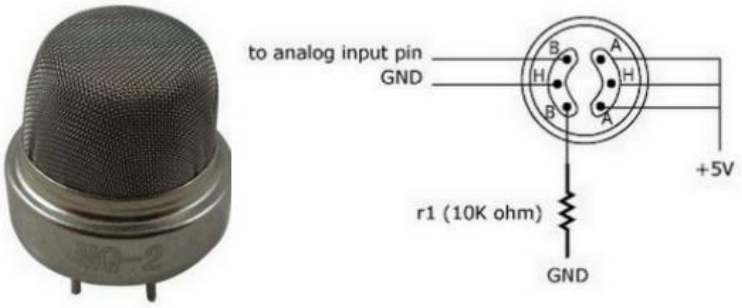


Рис. 1.

В газоаналізатор вбудований нагрівальний елемент, який необхідний для хімічної реакції. Тому під час роботи сенсор буде гарячим, це нормально. Для отримання стабільних показань новий сенсор необхідно один раз прогріти (залишити включеним) протягом 24 годин. Після цього стабілізація після включення буде займати близько хвилини.

Покази сенсора схильні до впливу температури і вологості навколишнього повітря. Тому в разі використання датчика газу в мінливому середовищі, при необхідності отримання точних показань, знадобиться реалізувати компенсацію цих параметрів. Схема вмикання датчика представлена на рис. 2.

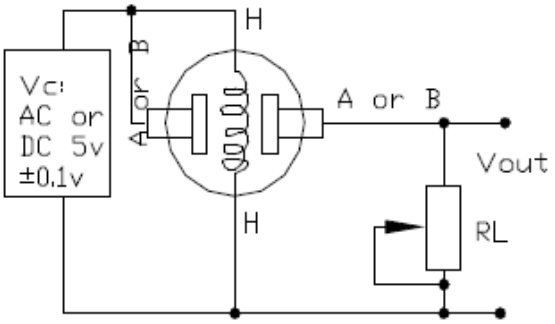


Рис. 2.

Характеристики датчика газу:
– Напруга живлення: 5 В;

– Споживаний струм: 160 мА.
Діапазон вимірювань:
– Пропан: 0,2 – 5 проміле;
– Бутан: 0,3 – 5 проміле;
– Метан: 5 – 2 проміле;
– Водород: 0,3 – 5 проміле.

Рис. 3 допомагає зрозуміти принцип роботи датчика. Напруга на виході датчика залежить від рівня концентрації газу в атмосфері.

Рис. 3.

Аналізуючи сигнал з виходу датчика можна зробити висновки про наявність небезпечної концентрації диму або відповідного газу у повітрі.

Приклад. Програма, структура якої представлена на рис. 4, виконує аналого-цифрове перетворення сигналів з датчиків різних газів, що підключені до входів АЦП мікроконтролера, та перетворює виміряне значення з апаратних одиниць в одиниці вимірювання концентрації (ppm). Виміряні значення передаються у текстовому вигляді по універсальному асинхронному приймачу/передавачу.

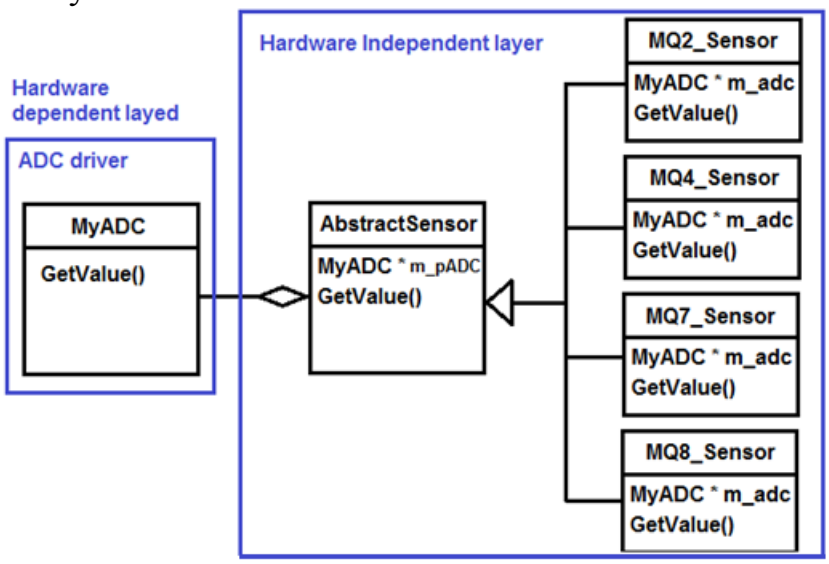


Рис. 4.

Принципову схему системи представлено на рис. 5.

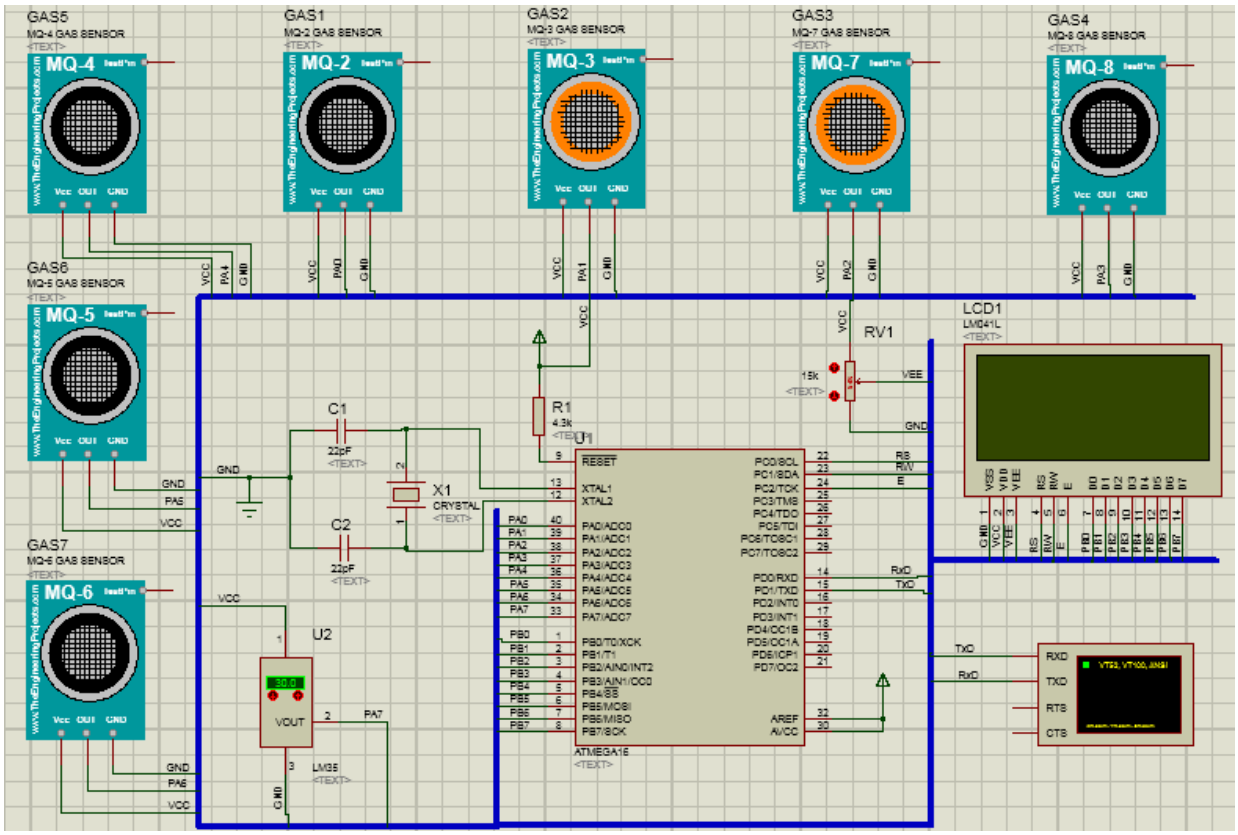


Рис. 5.

В програмі необхідно реалізувати декілька класів для роботи з датчиками. Однак всі датчики, що використовуються в цій системі збору даних, фактично однакові. Відмінність полягає у діапазоні вимірюваних концентрацій. Тому природнім в даному випадку є використання успадковування для реалізації ієрархії класів. При цьому в клас «абстрактного» датчика слід помістити всю функціональність, що не залежить від конкретного датчика, а в похідних класах реалізувати ці відмінності. Текст програми має вигляд:

```
#define F_CPU 8000000

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

#define MAX_STRING_LENGTH 16

class MyADC
{
public:
MyADC();
int GetValue(char channel);
};

MyADC::MyADC()
{
ADCSRA = 0b10000111;
}
```

```
int MyADC::GetValue(char channel)
{
char dh;
char dl;
int result;

ADMUX = channel;
ADCSRA |= 0b01000000;       // ADCSRA |= (1 << ADSC);
while((ADCSRA & (1 << ADIF)) == 0); // while(!(ADCSRA & (1 << ADIF));

dl = ADCL;
dh = ADCH;

result = dh;
result = result << 8;
result = result | dl;
return result;
}
```

Реалізація класу абстрактного датчика наведена нижче.

```
class AbstractSensor
{
protected:
char m_channel;
int m_value;
MyADC *m_pADC;

public:
AbstractSensor(char channel, MyADC * pADC);
int GetValue();
};
```

Конструктор класу приймає два аргументи – номер каналу АЦП, до якого підключено датчик, та вказівник на об’єкт для роботи з аналого-цифровим перетворювачем. Ці параметри конструктора зберігаються у відповідних полях класу.

```
AbstractSensor::AbstractSensor(char channel, MyADC * pADC)
{
m_channel = channel;
m_pADC = pADC;
}
```

Функція, що виконує аналого-цифрове перетворення сигналу з датчику в абстрактному класі не містить реалізації, адже для кожного окремого датчика коефіцієнти перетворення апаратних одиниць в фізичні будуть відрізнятися. Тому ця функція має бути реалізована в похідних класах для кожного конкретного датчика.

```
int AbstractSensor::GetValue()
{
//TODO: add code in child classes
return 0;
}
```

Клас для роботи з датчиком MQ2 успадковується від класу абстактного датчика. Оскільки в конструкторі класу відбувається лише копіювання переданих

аргументів в поля класу, що є спільним для всіх похідних класів, тому в конструкторі цього класу викликається конструктор базового класу, і в нього передаються відповідні аргументи.

```
class MQ2Sensor : AbstractSensor
{
public:
MQ2Sensor(char channel, MyADC *pADC) : AbstractSensor(channel, pADC)
{

};

int GetValue();
};

int MQ2Sensor::GetValue()
{
m_value = m_pADC->GetValue(m_channel);
m_value = 100 + (int)19900 * (double)m_value / 1023;
return m_value;
}
```

Реалізація класу для роботи з датчиком MQ3 наведена далі. Відмінності між класами полягають лише в реалізації функції GetValue(), в якій використовуються унікальні для кожного датчику коефіцієнти для перетворення апаратних одиниць в фізичні.

```
class MQ3Sensor : AbstractSensor
{
public:
MQ3Sensor(char channel, MyADC *pADC) : AbstractSensor(channel, pADC)
{

};

int GetValue();
};

int MQ3Sensor::GetValue()
{
m_value = m_pADC->GetValue(m_channel);
m_value = 500 + (int)9500 * (double)m_value / 1023;
return m_value;
}
```

Реалізація класу для роботи з датчиком MQ4 має вигляд:

```
class MQ4Sensor : AbstractSensor
{
public:
MQ4Sensor(char channel, MyADC *pADC) : AbstractSensor(channel, pADC)
{

};

int GetValue();
};

int MQ4Sensor::GetValue()
{
```

```
m_value = m_pADC->GetValue(m_channel);
m_value = 200 + (int)9800 * (double)m_value / 1023;
return m_value;
}
```

Реалізація класу для роботи з датчиком MQ5:

```
class MQ5Sensor : AbstractSensor
{
public:
MQ5Sensor(char channel, MyADC *pADC) : AbstractSensor(channel, pADC)
{

};

int GetValue();
};

int MQ5Sensor::GetValue()
{
m_value = m_pADC->GetValue(m_channel);
m_value = 200 + (int)9800 * (double)m_value / 1023;
return m_value;
}
```

Реалізація класу для роботи з датчиком MQ6:

```
class MQ6Sensor : AbstractSensor
{
public:
MQ6Sensor(char channel, MyADC *pADC) : AbstractSensor(channel, pADC)
{

};

int GetValue();
};

int MQ6Sensor::GetValue()
{
m_value = m_pADC->GetValue(m_channel);
m_value = 200 + (int)9800 * (double)m_value / 1023;
return m_value;
}
```

Реалізація класу для роботи з датчиком MQ7:

```
class MQ7Sensor : AbstractSensor
{
public:
MQ7Sensor(char channel, MyADC *pADC) : AbstractSensor(channel, pADC)
{

};

int GetValue();
};

int MQ7Sensor::GetValue()
{
m_value = m_pADC->GetValue(m_channel);
m_value = 20 + (int)1980 * (double)m_value / 1023;
```

```
return m_value;
}
```

Реалізація класу для роботи з датчиком MQ8:

```
class MQ8Sensor : AbstractSensor
{
public:
MQ8Sensor(char channel, MyADC *pADC) : AbstractSensor(channel, pADC)
{

};

int GetValue();
};

int MQ8Sensor::GetValue()
{
m_value = m_pADC->GetValue(m_channel);
m_value = 100 + (int)9900 * (double)m_value / 1023;
return m_value;
}

class TemperatureSensor : AbstractSensor
{
public:
TemperatureSensor(char channel, MyADC *pADC) : AbstractSensor(channel, pADC)
{

};

int GetValue();
};

int TemperatureSensor::GetValue()
{
m_value = m_pADC->GetValue(m_channel);
m_value = (int)100*((double)m_value *5.0 / 1023);
return m_value;
}

class MyUART
{
public:
MyUART();
void SendString(char *str);
};

MyUART::MyUART()
{
UCSRA = 0;
UCSRB = 0b00011000;
UCSRC = 0b00000110;
UBRRH = 0;
UBRRL = 51;
};

void MyUART::SendString(char *str)
{
int size = strlen(str);
if(size > MAX_STRING_LENGTH)
{
```

```
        size = MAX_STRING_LENGTH;
}

for(int i=0; i < size; i++)
{
        UDR = str[i];
        while (!( UCSRA & (1<<UDRE)));
}
}
```

В головній функції програми виконується створення екземплярів реалізованих класів та у циклі з періодом 1 секунда відбувається опитування всіх датчиків по черзі з виведенням результатів вимірювання у текстовому вигляді у вікно віртуального термінала. Це досягається шляхом передавання даних по універсальному асинхронному приймачу/передавачу, драйвер для роботи з яким було розроблено раніше.

```
int main(void)
{
    /* Replace with your application code */

MyADC myADC;
MyUART myUART;

char str[MAX_STRING_LENGTH];
int value;

MQ2Sensor mq2_sensor(0, &myADC);
MQ3Sensor mq3_sensor(1, &myADC);
MQ3Sensor mq4_sensor(4, &myADC);
MQ3Sensor mq5_sensor(5, &myADC);
MQ3Sensor mq6_sensor(6, &myADC);
MQ7Sensor mq7_sensor(2, &myADC);
MQ8Sensor mq8_sensor(3, &myADC);
TemperatureSensor temp_sensor(7, &myADC);

    while (1)
    {
       value = mq2_sensor.GetValue();
       sprintf(str, "C3H8: %d\r\n", value);
       myUART.SendString(str);

       value = mq3_sensor.GetValue();
       sprintf(str, "C2H5OH: %d\r\n", value);
       myUART.SendString(str);

       value = mq4_sensor.GetValue();
       sprintf(str, "CH4: %d\r\n", value);
       myUART.SendString(str);

       value = mq5_sensor.GetValue();
       sprintf(str, "NH3: %d\r\n", value);
       myUART.SendString(str);

       value = mq6_sensor.GetValue();
       sprintf(str, "CO2: %d\r\n", value);
       myUART.SendString(str);

       value = mq7_sensor.GetValue();
       sprintf(str, "CO: %d\r\n", value);
```

```
        myUART.SendString(str);

        value = mq8_sensor.GetValue();
        sprintf(str, "H2: %d\r\n", value);
        myUART.SendString(str);

        value = temp_sensor.GetValue();
        sprintf(str, "T: %d C\r\n", value);
        myUART.SendString(str);

        sprintf(str, "============\r\n");
        myUART.SendString(str);

        _delay_ms(1000);
    }
}
```

Результати моделювання процесу роботи мікропроцесорної системи збору даних представлено на рис. 5.

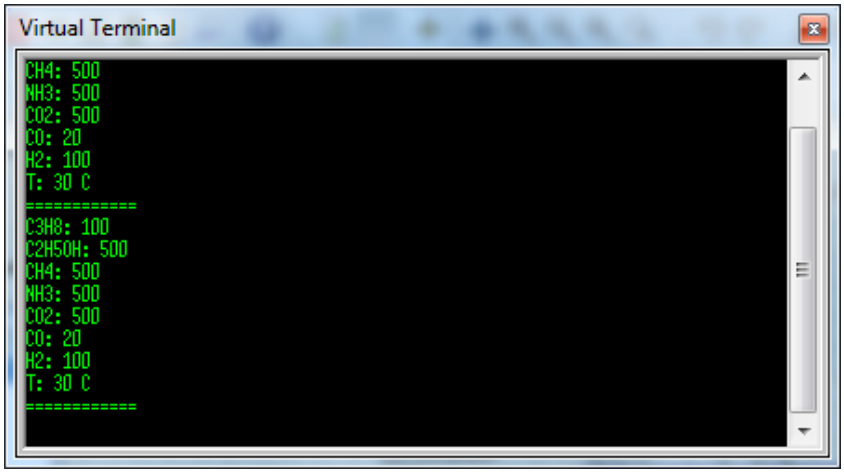


Рис. 5.

Аналіз отриманих результатів дозволяє зробити висновок про коректність роботи програми.

Завдання для самостійного виконання:

1. Додати в програму клас для роботи з алфавітно-цифровим РКІ та вивести виміряне значення концентарції газів на екран РКІ. Схема моделі системи керування, що розглянута в прикладі, розміщено на Google-диску (файл Proteus_model_Lection_10.pdsprj). Моделі датчиків газу можна завантажити з сайту за посиланням:

https://www.theengineeringprojects.com/2016/05/gas-sensor-library-proteus.html

2. Змінити протокол передавання даних по UART наступним чином. Довжина пакету даних має складати 20 байт. Структура пакету даних наступна:

Байт 1	**Байт 2**	**Байт 3**	**Байт 4**	**Байт 5**
заголовок пакету, символ ‘#’	адреса пристрою, діапазон 1-255	датчик MQ-2, старший байт	датчик MQ-2, молодший байт	датчик MQ-3, старший байт

Байт 6	Байт 7	Байт 8	Байт 9	Байт 10
датчик MQ-3, молодший байт	датчик MQ-4, старший байт	датчик MQ-4, молодший байт	датчик MQ-5, старший байт	датчик MQ-5, молодший байт

Байт 11	Байт 12	Байт 13	Байт 14	Байт 15
датчик MQ-6, старший байт	датчик MQ-6, молодший байт	датчик MQ-7, старший байт	датчик MQ-7, молодший байт	датчик MQ-8, старший байт

Байт 16	Байт 17	Байт 18	Байт 19	Байт 20
датчик MQ-8, молодший байт	датчик температ., старший байт	датчик температ., молодший байт	Контрольна сума, LRC	Кінець пакету, символ ‘#’

Перевірити роботу програми за допомогою віртуального терміналу, змінивши режим відображення на шістнадцятирічні значення.

3. Оформити звіт, в якому навести розроблені програми. Завантажити звіт на гугл-диск.

Практична робота 11.

Тема: Вивчення принципів роботи з інтерфейсом I2C. Розробка класів для роботи з комунікаційним інтерфейсом I2C та інтерфейсом UART. Організація взаємодії мікроконтролера та цифрового датчика температури DS1621 з використанням послідовного синхронного інтерфейсу I2C.

Шина I2C широко використовується в побутовій електроніці, передачі даних і промисловій електроніці. Розроблена фірмою Philips проста двонапрямлена 2-провідна шина спочатку для ефективного керування та взаємодії різних блоків телевізорів, стала застосовуватися для зв'язку між собою однокристальних мікроконтролерів, РКІ-індикаторів, мікросхем пам'яті, аналого-цифрових і цифро-аналогових перетворювачів, мікросхем реального часу та ін.

Для здійснення процесу обміну інформацією по шині I2C використовуються всього два сигнали – лінія даних SDA та лінія синхронізації SCL. Для забезпечення реалізації двонапрямленості шини без застосування складних арбітрів шини вихідні каскади пристроїв, підключених до шини, мають відкритий стік або відкритий колектор для забезпечення функції монтажного "І". Як показано на рис. 1, обидві лінії шини підключені до живлення через резистори.

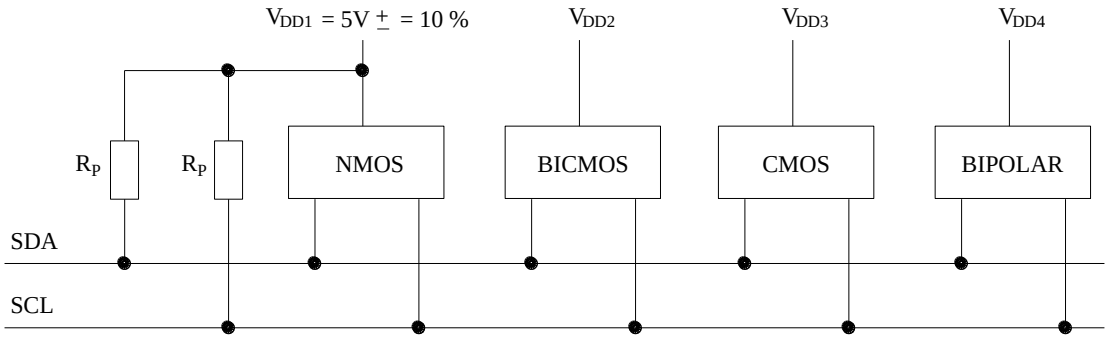


Рис. 1.

Всі абоненти шини діляться на два класи – «Master» та «Slave». Пристрій «Master» генерує тактовий сигнал (SCL), і тому є ведучим. Він може самостійно виходити на шину й адресуватися до будь-якого «Slave» – пристрою з метою передачі або прийому інформації. Всі «Slave» – пристрої «слухають» шину на предмет виявлення власної адреси, і, розпізнавши її, виконують вказані дії. Крім того, можливий режим «Multi-master» – режим, коли на шині встановлено кілька ведучих («Master») пристроїв, які або спільно розділяють загальні «Slave» – пристрої, або почергово є ведучими пристроями, коли самі ініціюють обмін інформацією.

Мікросхема DS1621 – це цифровий термометр і термостат. Прилад є закінченим конвертером класу «температура – цифра» і не вимагає при використанні ніяких зовнішніх компонентів. Негативні значення температури представлені в коді з доповненням до 2-х. Прилад складається з п'яти основних компонентів:

– прецизійний датчик температури;
– аналого-цифровий перетворювач;
– електроніка двохпровідного інтерфейсу;
– регістр даних;
– компаратор термостата.

Ведучий шини може періодично зчитувати значення з температурного регістра, що містить результат попереднього перетворення.

Перший байт, переданий ведучим пристроєм після умови «Старт», містить адресу пристрою. Для DS1621 це значення 1001. Далі передаються 3 біти вибору пристрою A2, A1, A0. Якщо використовується тільки один пристрій на шині I2C, то значення цих біт може бути 0 за умови, що однойменні виводи мікросхеми заземлені. Останній біт – біт зчитування/запису – визначає напрямок передачі даних. Після цього команда, що визначає подальші дії по організації обміну даними. Набір команд для DS121 приводиться в табл. 1.

Таблиця 1.

Команда	Опис	Код команди
Зчитування температури	Зчитує останнє значення температури з регістра температури	0xAA
Зчитування лічильника	Зчитує значення відліку, що залишається в лічильнику	0xA8
Зчитування TH	Зчитування збереженого значення межи високої температури з регістра TH, R/W=0	0xA1
Зчитування TL	Зчитування збереженого значення межи низької температури з регістра TL, R/W=0	0xA2
Запис TH	Запис значення межи високої , температури в регістр TH, R/W=1	0xA1
Запис TL	Запис значення межи низької температури в регістр TL, R/W=1	0xA2
Запис конфігурації	Запис даних конфігурації в регістр конфігурації, R/W=1	0xAC
Зчитування конфігурації	Зчитування даних з регістра конфігурації, R/W=0	0xAC

Почати перетворення	Початок перетворення температури, R/W=0	0xEE
Зупинити перетворення	Зупинка перетворення температури в безперервному режимі, R/W=0	0x22

У табл. 2 представлена відповідність значення температури цифровому коду.

Таблиця 2.

Температура, ^{0}C	Код (BIN)	Код (HEX)
+ 125	01111101 00000000	7B00h
+ 25	00011001 00000000	1900h
+ 0,5	00000001 00000000	0080h
0	00000000 00000000	0000h
– 0,5	11111111 10000000	FF80h
– 25	11100111 00000000	E700h
– 55	11001001 00000000	C900h

В наступному прикладі розглянуто програму для мікроконтролера ATMega16, що виконує зчитування температури з датчика DS1621 по інтерфейсу I2C.

Приклад. Програма виконує зчитування значення температури з датчика DS1621 кожну секунду. Виміряні значення передаються у текстовому вигляді по універсальному асинхронному приймачу/передавачу.

Принципову схему системи представлено на рис. 2.

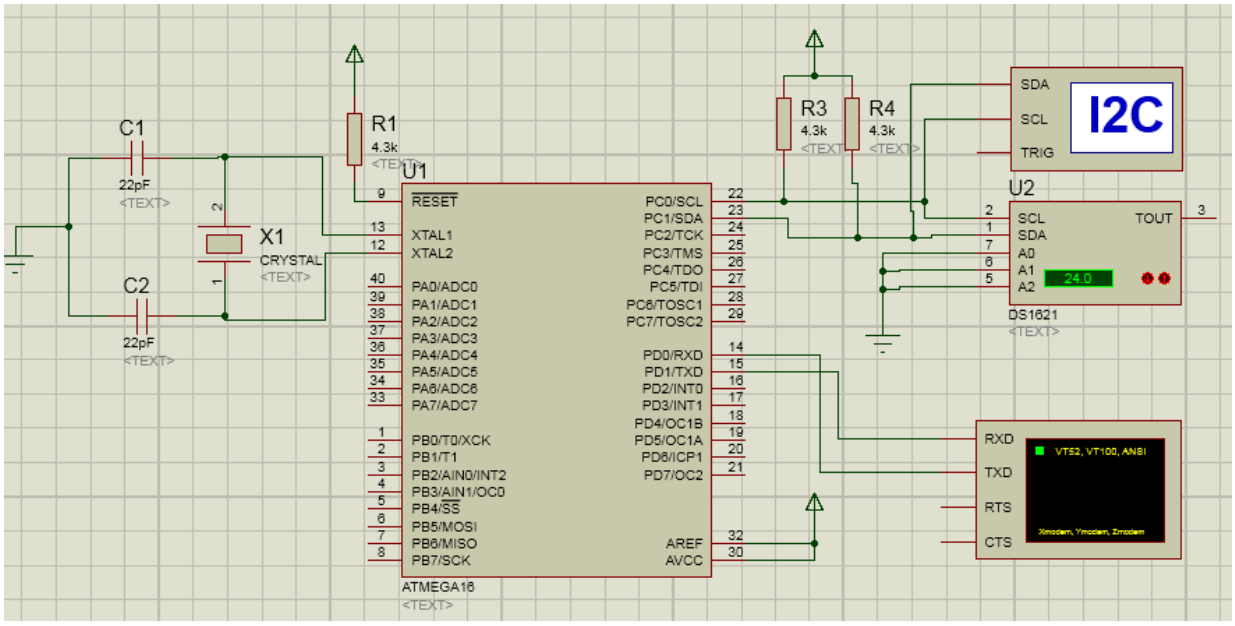


Рис. 2.

Текст програми:

```
#define F_CPU 8000000
#define MAX_STRING_LENGTH 32

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

```
#include <string.h>
#include <stdio.h>
```

Клас для роботи з інтерфейсом I2C містить поля для зберігання номерів виводів порта, до яких підключено датчик, та адриси самого порта. Також у класі оголошено додаткові функці для керування лініями SDA та SCL – установки високих та низьких рівнів сигналів на цих лініях. Для керування початком процесу передавання даних, формуванням умови завершення процесу обміну даними та інших операції також реалізовано декілька службових функцій. Ці функції знаходяться у секції private, тому не доступні для виклику іззовні класу. Вони можуть бут використані лише іншими членами-функціями цього класу.

```
class I2C
{
private:
volatile unsigned char *m_port;
char m_sdaPin;
char m_sclPin;

void SetSDA();
void SetSCL();
void ResetSDA();
void ResetSCL();
void SetSDAOut();
void SetSDAIn();
void SetSCLOut();
void SetSCLIn();
void delay_us(int us);

public:
I2C(volatile unsigned char * port, char sdaPin, char sclPin);
char IsBusy();
void Start();
void Stop();
char SendByte(char d);
char ReadByte(unsigned char acknowledge);
};
```

Реалізація конструктора класу. В конструктор класу передаються три параметри – адреса порта, до якого підключено датчик, та номери виводів.

```
I2C::I2C(volatile unsigned char * port, char sdaPin, char sclPin)
{
m_port = port;
m_sdaPin = sdaPin;
m_sclPin = sclPin;
}
```

Функція-член класу для установки високого рівня сигналу на лінії SDA.

```
void I2C::SetSDA()
{
*m_port |= (1<<m_sdaPin);
}
```

Функція-член класу для установки високого рівня сигналу на лінії SCL.

```
void I2C::SetSCL()
```

```
{
*m_port |= (1<<m_sclPin);
}
```

Функція-член класу для установки низького рівня сигналу на лінії SDA.

```
void I2C::ResetSDA()
{
*m_port &= ~(1<<m_sdaPin);
}
```

Функція-член класу для установки низького рівня сигналу на лінії SCL.

```
void I2C::ResetSCL()
{
*m_port &= ~(1<<m_sclPin);
}
```

Функція-член класу для налаштування виводу мікроконтроллера, що керує лінією SDA, на вихід.

```
void I2C::SetSDAOut()
{
(*(m_port-1)) |= (1<<m_sdaPin); // SDA pin in OUT mode - data can be send
}
```

Функція-член класу для налаштування виводу мікроконтроллера, що керує лінією SDA, на вхід.

```
void I2C::SetSDAIn()
{
(*(m_port-1)) &= ~(1<<m_sdaPin); // SDA pin in IN mode - data can be received
}
```

Функція-член класу для налаштування виводу мікроконтроллера, що керує лінією SCL, на вихід.

```
void I2C::SetSCLOut()
{
(*(m_port-1)) |= (1<<m_sclPin); // SDA pin in OUT mode - data can be send
}
```

Функція-член класу для налаштування виводу мікроконтроллера, що керує лінією SCL, на вхід.

```
void I2C::SetSCLIn()
{
(*(m_port-1)) &= ~(1<<m_sclPin); // SDA pin in IN mode - data can be received
}
```

Функція формування часової затримки (мікросекунди).

```
void I2C::delay_us(int us)
{
int c1;                          // оголошення додаткової змінної
for(c1=0;c1<us;c1++) __asm("nop");// у циклі виконується «порожня» операція
}
```

Функція, що використовується для перевірки, чи лінія I2C вільна.

```
char I2C::IsBusy()
{
char st;                    // додаткова змінна
SetSCLIn();                 // налаштування на вхід
SetSDAIn();
if(((*(m_port-2)) & ((1<<m_sclPin) | (1<<m_sdaPin))) != ((1<<m_sclPin) | (1<<m_sdaPin))) st=1;
       // перевірка стану ліній
else st=0;                  // функція повертає 0 якщо шина вільна
return st;                  // інакше повертається значення 1
}
```

Функція формування ознаки початку процесу передавання даних – при високому рівні на лінії SCL відбувається перехід сигналу з високого рівня в низький на лінії SDA.

```
void I2C::Start()
{
SetSDAOut();
SetSCLOut();
SetSDA();
SetSCL();
delay_us(2);          // затримка
ResetSDA();           // скидання сигналу SDA
delay_us(2);          // затримка
ResetSCL();           // скидання сигналу SCL
delay_us(2);          // затримка
}
```

Функція формування ознаки завершення процесу передавання даних.

```
void I2C::Stop()
{
SetSDAOut();
SetSCLOut();
ResetSDA();
SetSCL();
delay_us(2);
SetSDA();
delay_us(2);
}
```

Функція для відправки байту даних.

```
char I2C::SendByte(char d)
{
  unsigned char i=8,acknowledge=0; // додаткові змінні
  do {                             // у циклі виконується посилка 8 біт
        if((d & 0x80)!=0) SetSDA();      // установка сигналу SDA якщо
        // старший біт дорівнює 1
        else ResetSDA();           // інакше скидання сигналу SDA
        d = (d << 1);              // зсув переданого байта на 1 розряд ліворуч
        delay_us(2);               // формування затримки
        // формування тактового імпульсу
        SetSCL();                  // установка сигналу SCL
        delay_us(2);               // затримка
        ResetSCL();                // скидання сигналу SCL
        delay_us(2);               // затримка
  }
  while (--i);                           // цикл виконується 8 разів
```

```
  // формування тактового імпульсу для біта підтвердження
  SetSDAIn();// переведення лінії SDA у третій стан
  ResetSDA();                       // скидання сигналу SDA
  SetSCL();                         // установка сигналу SCL
  delay_us(2);                      // затримка
  if (((*(m_port-2)) & (1<<m_sclPin)) == 0) acknowledge=1; // якщо на лінії SCL низький
  // рівень, біт підтвердження прийнятий
  ResetSCL();                               // скидання сигналу SCL
  delay_us(2);                              // затримка
  SetSCLOut();
  SetSDAOut();
  return acknowledge;               // повертається біт підтвердження

}
```

Функція для зчитування байту даних.

```
char I2C::ReadByte(unsigned char acknowledge)
{
  unsigned char i=8,i2c_data=0;    // додаткові змінні
  char d;
  SetSDAIn();               // переведення лінії SDA у третій стан
  do {
        i2c_data<<=1;                       // зсув на 1 розряд ліворуч
        SetSCL();                 // установка сигналу SCL
        delay_us(2);                        // затримка
        d=(*(m_port-2)&(1<<m_sdaPin));          // зчитування стану порту
        d=(d>>m_sdaPin)&0x01;     // виділення 0 біта
        i2c_data|=d;                        // копіювання 0 біта в змінну
        ResetSCL();               // скидання сигналу SCL
        delay_us(2);                        // затримка
  }
  while (--i);                              // цикл виконується 8 разів
  if (acknowledge)                // якщо необхідний біт підтвердження
  {
        SetSCLOut();
        SetSDAOut();
        ResetSDA();               // скидання сигналу SDA
  }
  // формування тактового імпульсу
  SetSCL();                         // установка сигналу SCL
  delay_us(2);                      // затримка
  ResetSCL();               // скидання сигналу SCL
  SetSDAIn();               // переведення лінії SDA у третій стан
  return i2c_data;          // функція повертає прийнятий байт

}

class MyUART
{
public:
MyUART();
void SendString(char *str);
};

MyUART::MyUART()
{
UCSRA = 0;
UCSRB = 0b00011000;
UCSRC = 0b00000110;
UBRRH = 0;
UBRRL = 51;
```

```
};

void MyUART::SendString(char *str)
{
int size = strlen(str);
if(size > MAX_STRING_LENGTH)
{
	size = MAX_STRING_LENGTH;
}

for(int i=0; i < size; i++)
{
	UDR = str[i];
	while (!( UCSRA & (1<<UDRE)));
}
}
```

Головна функція програми (точка входження). В цій функції відбувається створення об’єкту для роботи з датчиком, підключеним по інтерфейсу I2C.

```
int main(void)
{
    I2C temp(&PORTC, 1, 0);
    int c1=0,c2=0;          // оголошення додаткових змінних
    int t1, t2;
char str[MAX_STRING_LENGTH];

MyUART uart;

    while (1)
    {
       while (temp.IsBusy() != 0); // чекаємо поки лінія не звільниться

        temp.Start();              // функції формування сигналу початку передачі
        temp.SendByte(0b10010000); // відправлення байта 0x90 (адреса)
        temp.SendByte(0xEE);             // відправлення команди 0xEE
        temp.Stop();                     // формування сигналу закінчення передачі

        temp.Start();                    // початок передачі даних
        temp.SendByte(0b10010000);       // відправлення байта 0x90 (адреса)
        temp.SendByte(0xAA);             // відправлення команди 0xAA
        temp.Stop();                     // завершення передачі даних
        temp.Start();                    // повторний старт
        temp.SendByte(0b10010001); // відправлення команди зчитування
        c1=temp.ReadByte(1);             // зчитування старшого байта температури
        c2=temp.ReadByte(0);             // зчитування молодшого байта температури
        temp.Stop();                     // звільнення лінії I2C
        t1=c1/10;                        // формування значення температури (десятки)
        t2=c1-t1*10;                     // формування значення температури (одиниці)

       sprintf(str, "T=%d%d C\r\n", t1, t2);
       uart.SendString(str);

       _delay_ms(1000);
    }
}
```

Результат роботи програми представлено на рис. 3.

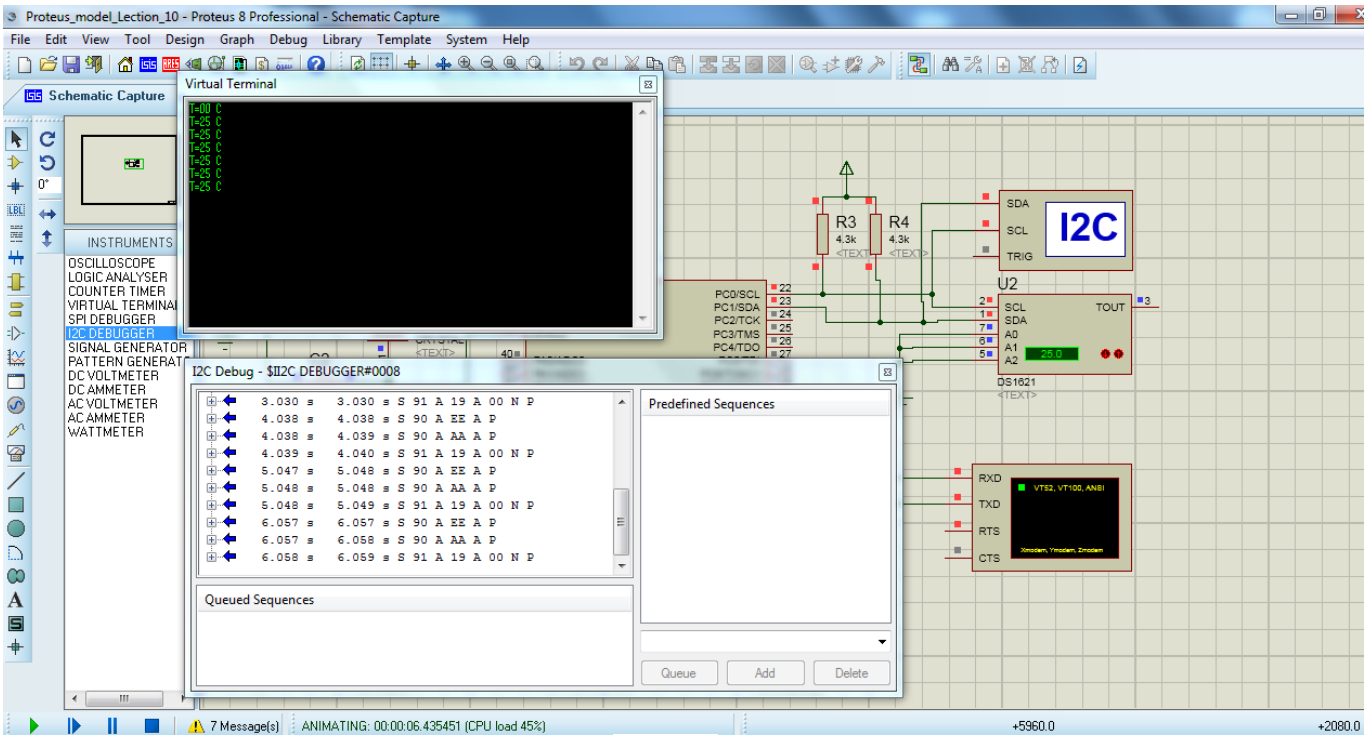


Рис. 3.

Аналіз відображених даних у вікні віртуального терміналу, а також надісланих та прийнятих по інтерфейсу I2C пакетів підтверджує коректність роботи програми.

Завдання для самостійного виконання:

1. Виконати рефакторинг програмного коду таким чином, щоб оформити процес опитування датчика температури у вигляді окремого класу (тобто всю функціональність, яка в прикладі знаходиться у функції main() та стосується опитування датчика температури, помістити в окремий клас). В результаті функція main() повинна виглядати наступним чином:

```
int main(void)
{
DS1621 sensor(&PORTC, 1, 0);
         int temperature;
char str[MAX_STRING_LENGTH];

MyUART uart;

    while (1)
    {
       temperature = sensor.ReadTemperature();

       sprintf(str, "T=%d C\r\n",temperature);
       uart.SendString(str);

       _delay_ms(1000);
    }
}
```

Проєкт має скомпілюватися і результат роботи програми повинен бути таким самим, як зараз в прикладі. Схема моделі системи керування, що

розглянута в прикладі, розміщено на Google-диску (файл Proteus_model_Lection_11.pdsprj).
Інформацію про інтерфейс I2C можна знайти в підручнику «В.М. Рябенький, О.О. Ушкаренко, В.С. Буряк. Схемотехніка електронних пристроїв та систем. Том 3. Мікропроцесорна техніка», Розділ 5, що знаходиться на Google-диску в папці «Книги».

2. Оформити звіт, в якому навести розроблену програму. Завантажити звіт на гугл-диск.

Практична робота 12.

Тема: Вивчення принципів організації міжпроцесорної взаємодії по послідовному синхронному інтерфейсу SPI. Розробка класу для роботи з інтерфейсом SPI.

Апаратна частина послідовного периферійного інтерфейсу представляє собою регістр зсуву, що послідовно передає біти даних в аналогічні інтерфейси інших пристроїв. Під час процесу передачі один із пристроїв працює як ведучий, забезпечуючи контроль над потоком даних, у той час як інші, ведені пристрої, приймають або передають дані під керуванням ведучого. Функції провідного пристрою інтерфейсу можуть передаватися від одного процесора іншому (мультимастерний протокол у порівнянні з одномастерним, коли тільки один процесор завжди працює в якості ведучого, а всі інші в якості ведених), і ведучий пристрій може вести передачу даних декільком веденим пристроям одночасно.
З'єднання між ведучим й веденим пристроями показане на рис. 1.

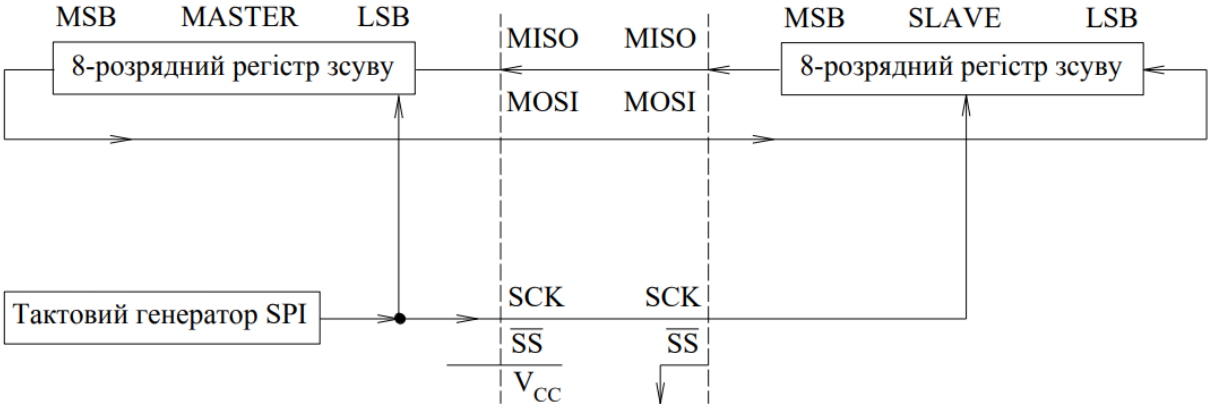


Рис. 1.

Два регістри зсуву ведучого і веденого можна розглядати як один рознесений 16-розрядний циклічний регістр зсуву. При зсуві даних із ведучого мікроконтролера у ведений одночасно відбувається зсув даних з веденого мікроконтролера у ведучий, тобто протягом одного циклу зсуву відбувається обмін даними між ведучим й веденим мікроконтролерами.

Вивід SCK є виходом тактового сигналу ведучого мікроконтролера і входом тактового сигналу веденого. При записі ведучим мікроконтролером даних у регістр SPI починає працювати тактовий генератор, і записані дані передаються з

виходу MOSI ведучого мікроконтролера на вхід MOSI веденого. Після передачі одного байта генератор зупиняється, встановлюючи прапор завершення передачі SPIF. Якщо в регістрі SPCR встановлений прапор дозволу переривання (SPIE), то відбудеться запит переривання. Вхід вибору веденого SS, для вибору індивідуального SPI пристрою в якості веденого, встановлюється на низький рівень. При установці високого рівня на виводі SS порт SPI деактивується. Режим ведучий/ведений може бути встановлений програмним способом установкою або очищенням біта MSTR у регістрі керування SPI.

При роботі SPI ведучим (біт MSTR регістра SPCR встановлений), користувач має можливість установити напрямок роботи виводу #SS. Якщо вивід #SS налаштований як вихід, то вивід є виводом загального призначення і він не активується системою SPI. Якщо ж вивід #SS налаштований як вхід, то для забезпечення роботи ведучого SPI він повинен утримуватися на високому рівні. Якщо в режимі ведучого вивід #SS є входом, і зовнішньою периферійною системою на нього поданий низький рівень, то SPI сприйме його як звернення іншого провідного SPI до себе, як до веденого. Щоб уникнути конфліктної ситуації на шині, система SPI виконує наступні дії:

1. Біт MSTR у регістрі SPCR очищається й SPI система стає веденою. Результатом цього є те, що виводи MOSI та SCK стають входами.

2. Встановлюється прапор SPIF регістра SPSR, і якщо дозволено переривання SPI, починається виконання підпрограми обробки переривання.

Якщо SPI працює в режимі веденого, то вивід #SS постійно працює на вхід. Якщо на вивід #SS поданий низький рівень, то SPI активується і вивід MISO стає виходом. Всі інші виводи є входами. Якщо вивід #SS утримується на високому рівні, то всі виводи є входами, SPI пасивний, і це означає, що він не буде одержувати вхідних даних.

Існує чотири варіанти комбінації фази й полярності SCK відносно послідовних даних, які встановлюються керуючими бітами CPHA та CPOL.

Після переходу SPI у режим веденого вивід SS завжди працює як вхід. У цьому випадку SPI активується, якщо на вхід SS подати низький рівень, а вивід MISO стає виходом. Всі інші виводи працюють як входи. Якщо на вхід SS подати високий рівень, то всі виводи стануть входами і SPI перейде в пасивний стан, у якому блокується прийом вхідних даних. Логіка SPI скидається, як тільки на вивід SS подається високий логічний рівень.

Опис регістрів SPI для контролерів AVR приводиться нижче.

Регістр SPCR – регістр керування:

Розряд	7	6	5	4	3	2	1	0
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
Зчитування /запис	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.
Початкове значення	0	0	0	0	0	0	0	0

SPIE – дозвіл переривання SPI.

SPE – SPI включений. Якщо встановлено цей біт, то виводи #SS, MISO, MOSI та SCK працюють як виводи SPI, інакше – як прості виводи порту.
DORD – напрямок передачі даних. Якщо цей біт встановлено, то передача починається з молодшого біта, якщо скинуто – зі старшого.
MSTR – якщо встановлено, контролер працює як Master, якщо скинуто – як Slave. Керується також виводом #SS, якщо він налаштований на вхід – при подачі "0" на #SS біт MSTR скидається.
CPOL – визначає рівень на виході SCK у режимі очікування – SCK = CPOL.
CPHA – якщо встановлено, передача і прийом біта відбуваються по спаду сигналу, якщо скинуто – по фронту.
SPR1, SPR0 – дільник тактової частоти (F – частота генератора контролера).

SPR1	SPR0	Частота
0	0	F/4
0	1	F/16
1	0	F/64
1	1	F/128

Регістр SPSR – регістр стану:

Розряд	7	6	5	4	3	2	1	0
	SPIF	WCOL	–	–	–	–	–	SPI2X
Зчитування /запис	Зч.	Зч.	Зч.	Зч.	Зч.	Зч.	Зч.	Зч./Зап.
Початкове значення	0	0	0	0	0	0	0	0

SPIF – прапор переривання по закінченню обміну даними. Встановлюється й скидається апаратно. Якщо біт SPIE у регістрі SPCR встановлений, і дозволене глобальне переривання, генерується сигнал переривання. Біт SPIF може бути очищений також при першому зчитуванні регістра стану.
WCOL – прапор помилки при записі. Встановлюється, коли прийнятий байт накладається на ще не зчитаний з регістра даних. Скидається зчитуванням регістра стану.
Біти 5..1 – зарезервовані біти. При зчитуванні завжди покажуть 0.
SPI2X – біт подвоєння швидкості SPI.

Регістр SPDR – регістр даних:

Розряд	7	6	5	4	3	2	1	0
	Ст. розр.							Мол. розр.
Зчитування /запис	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.	Зч. /Зап.
Початкове значення	x	x	x	x	x	x	x	x

Регістр даних SPI представляє собою регістр із можливістю зчитування/запису і призначений для пересилання даних. Запис у регістр SPDR ініціює передачу даних, зчитування регістра призводить до зчитування даних із регістра зсуву приймача. Передача та прийом тактуються по лінії SCK відповідно до конфігурації, що задана в регістрі керування. Після того, як байт переданий, тактування припиняється, у регістрі стану встановлюється прапор SPIF і викликається переривання по закінченню передачі, якщо воно дозволено. Запис у регістр даних передаючого контролера неможливий, поки не звільнений регістр зсуву SPI.

В SPI установка й передача даних відбувається по протилежних фронтах тактових імпульсів. Такий поділ операцій зсуву та передачі має певні переваги, дозволяючи уникати жорсткого часового режиму між цими двома операціями. Це дозволяє полегшити задачу синхронізації при розробці інтегральних схем або плат. Але, з іншого боку, через комбінації полярності й фази синхроімпульсів виникає чотири режими роботи, і ведучому пристрою доводиться налаштовуватися на той режим, що використовується веденим.

Приклад 1. В програмі реалізовано клас для роботи з інтерфейсом SPI. При цьому шляхом умовної компіляції виконується створення окремих .hex файлів для ведучого мікроконтролера та веденого мікроконтролера, що з'єднані між собою по інтерфейсу SPI. Ведучий контролер виконує опитування станів чотирьох дискретних кнопок, підключених до порту B, і передає інформацію про їх стан з періодом 200 мс до веденого мікроконтролера по інтерфейсу SPI. Ведений выкроконтролер выдображає отримане значення про стани кнопок на світлодіодах, підключених до порту C (при натисканні на кнопку загоряється відповідній світлодіод).

Принципову схему системи (файл Proteus_model_Lection_12a.pdsprj) представлено на рис. 2.

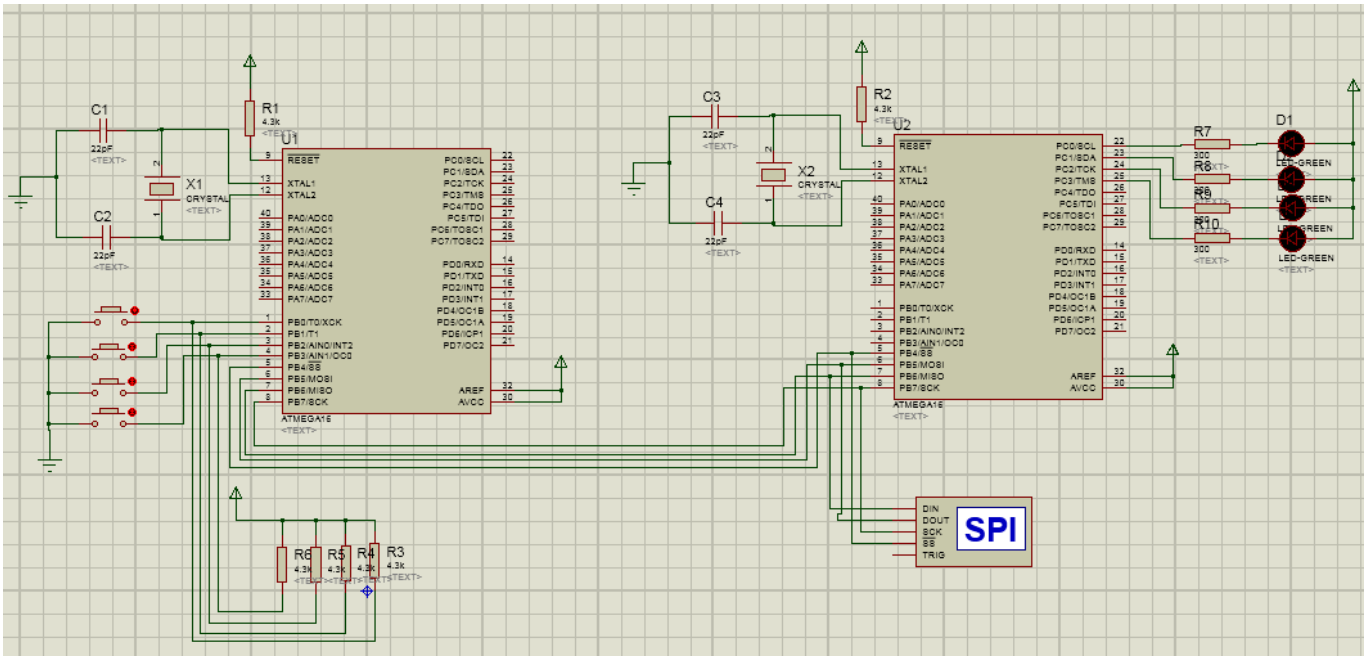


Рис. 2.

Текст програми:

```
#define F_CPU 8000000
#define _SPI_MASTER_

#define LOW 0
#define HIGH 1

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

class Port
{
volatile unsigned char * m_pPort;
public:
Port(volatile unsigned char * pPort, bool isOutput);
void Write(char data);
char Read();
void SetPinOut(char pinNum);
void SetPinIn(char pinNum);
void SetPin(char pinNum);
void ResetPin(char pinNum);
char GetPinState(char pinName);
};

Port::Port(volatile unsigned char * pPort, bool isOutput)
{
m_pPort = pPort;
if(isOutput)
{
	*(m_pPort-1) = 0xFF;
}
}


void Port::Write(char data)
{
*m_pPort = data;
}

char Port::Read()
{
char value;
value = *(m_pPort - 2);
return value;
}

void Port::SetPinOut(char pinNum)
{
*(m_pPort-1) |= (1 << pinNum);
}

void Port::SetPinIn(char pinNum)
{
*(m_pPort-1) &= ~(1 << pinNum);
}

void Port::SetPin(char pinNum)
{
*m_pPort |= (1 << pinNum);
```

```
}

void Port::ResetPin(char pinNum)
{
*m_pPort &= ~(1 << pinNum);
}

char Port::GetPinState(char pinName)
{
char state = 0;
state = (*(m_pPort - 2)) & (1 << pinName);
if(state != 0) return LOW;
else return HIGH;
}


class MySPI
{
public:
MySPI(bool isMaster);
char Send(char data);
char Receive(char data = 0);
};

MySPI::MySPI(bool isMaster)
{
if(isMaster)
{
	SPCR = 0b01011111;
}
else
{
	SPCR = 0b01001111;
}
}

char MySPI::Send(char data)
{
char received_data = 0;
SPDR = data;
while((SPSR & (1 << SPIF)) == 0);
received_data = SPDR;
return received_data;
}

char MySPI::Receive(char data)
{
char received_data;
while((SPSR & (1 << SPIF)) == 0);
received_data = SPDR;
SPDR = data;
return received_data;
}

#ifdef _SPI_MASTER_

int main(void)
{
char data;
Port buttons(&PORTB, false);
MySPI spi_master(true);
buttons.SetPinOut(4);
buttons.SetPinOut(5);
```

```
buttons.SetPinOut(7);

    while (1)
    {
       data = buttons.Read() & 0x0F;
       spi_master.Send(data);
       _delay_ms(200);
    }
}

#else

int main(void)
{
char data;
Port leds(&PORTC, true);
MySPI spi_slave(false);

while (1)
{
       data = spi_slave.Receive();
       leds.Write(data);
}
}

#endif
```

Після запуску процесу моделювання у вікні аналізатора протоколу (рис. 3) можна побачити вміст пакетів, що передаються від ведучого мікроконтролера веденому і навпаки.

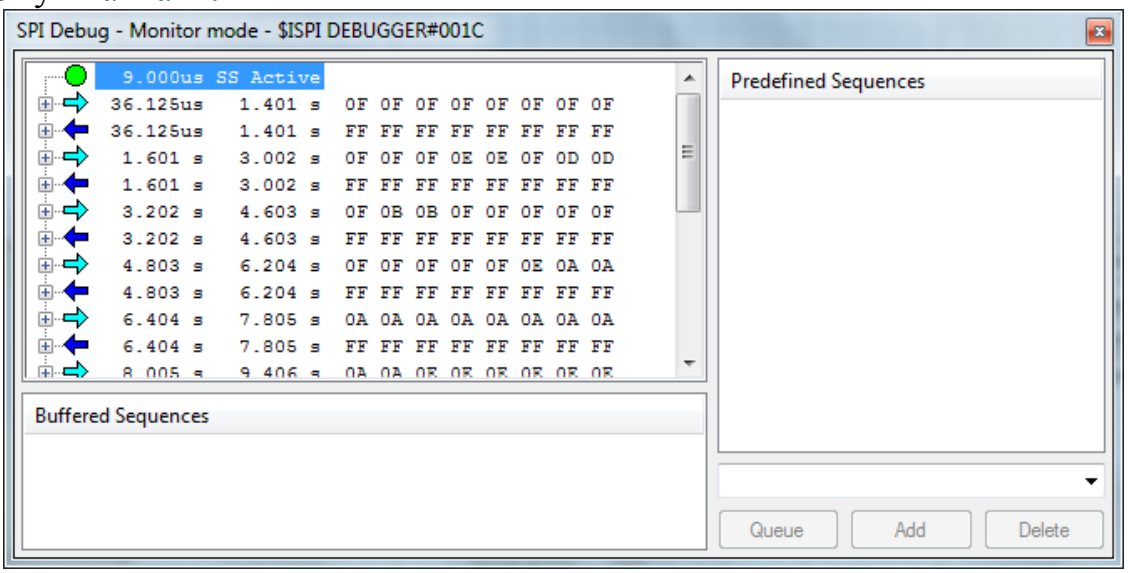


Рис. 3.

Для того, щоб аналізатор протоколу відображав коректні дані, налаштування повинні бути такими, як показано на рис. 4.

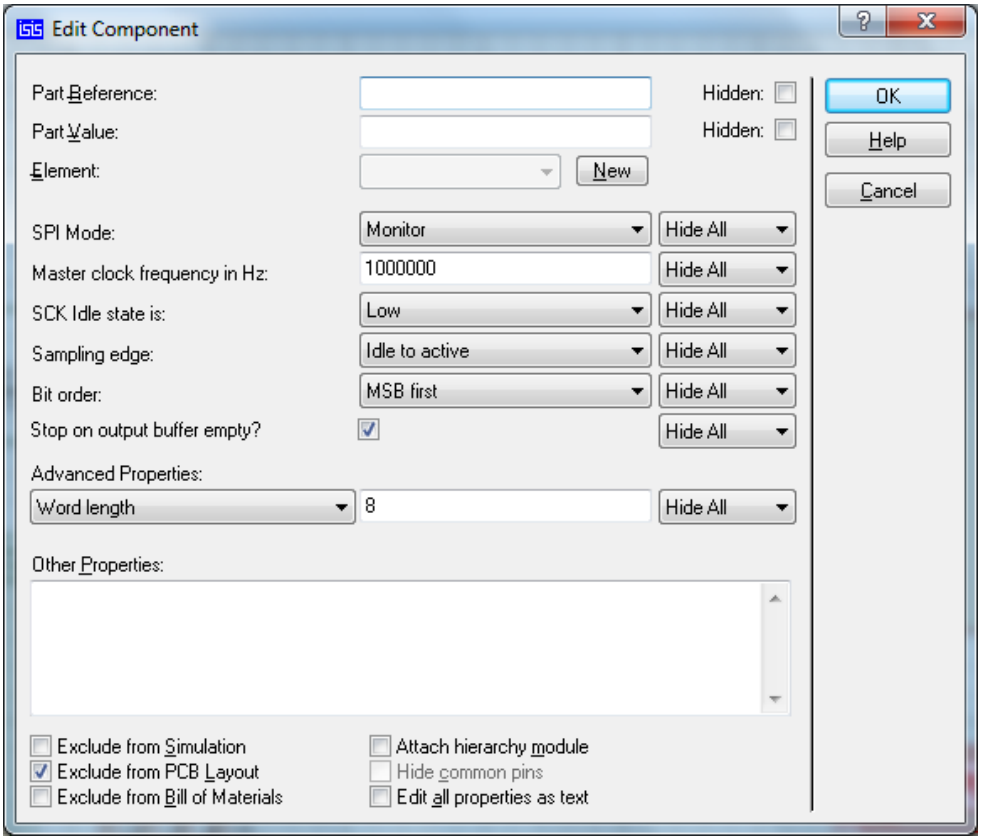


Рис. 4.

Приклад 2. В програмі реалізовано клас для роботи з мікросхемою MAX6902, яка представляє собою годинник реального часу (RTC) з інтерфейсом SPI. Кожну секунду виконується зчитування поточного часу з мікросхеми-годинника, виконується обробка зчитаних даних для їх перетворення у текстовий рядок та відправка по UART для відображення у вікні терміналу.

Принципова схема системи (файл Proteus_model_Lection_12b.pdsprj) представлена на рис. 5.

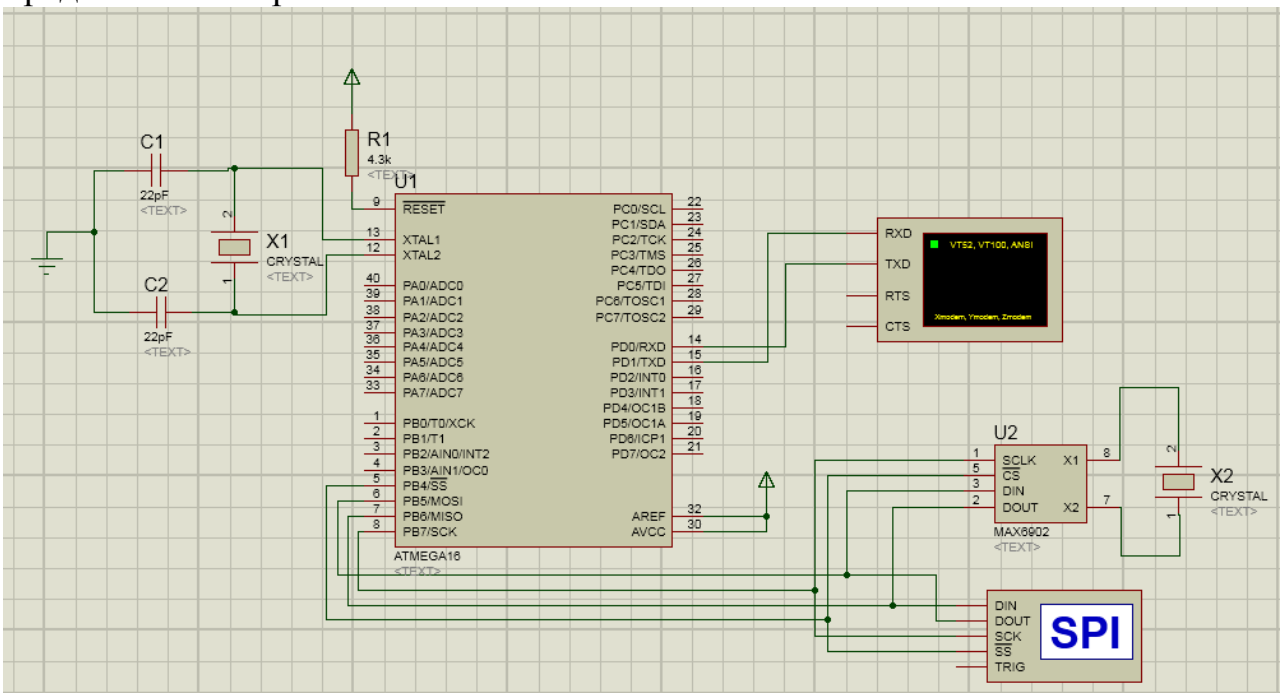


Рис. 5.

На рис. 6 показано структуру адресного простору годинника реального часу. Зчитування поточного часу відбувається шляхом відправки команди 0xBF, після чого виконується зчитування 8 регістрів, що містять інформацію про дату та час.

	REGISTER ADDRESS								REGISTER DEFINITION								
FUNCTION CLOCK		A7	A6	A5	A4	A3	A2	A1	A0	VALUE	D7	D6	D5	D4	D3	D2	D1	D0
SECONDS	RD	0	0	0	0	0	0	0	1	00-59	0	10 SEC			1 SEC			
	/W									*POR STATE	0	0	0	0	0	0	0	0
MINUTES	RD	0	0	0	0	0	0	1	1	00-59	ALM OUT	10 MIN			1 MIN			
	/W									*POR STATE	0	0	0	0	0	0	0	0
HOURS	RD	0	0	0	0	0	1	0	1	00-23	12/24	0	10 HR	10 HR	1 HR			
	/W									01-12	1/0		A/P 0/1					
										*POR STATE	0	0	0	0	0	0	0	0
DATE	RD	0	0	0	0	0	1	1	1	01-28/29 01-30 0-31	0	0	10 DATE		1 DATE			
	/W									*POR STATE	0	0	0	0	0	0	0	1
MONTH	RD	0	0	0	0	1	0	0	1	01-12	0	0	0	10M	1 MONTH			
	/W									*POR STATE	0	0	0	0	0	0	0	1
DAY	RD	0	0	0	0	1	0	1	1	01-07	0	0	0	0	0	WEEK DAY		
	/W									*POR STATE	0	0	0	0	0	0	0	1
YEAR	RD	0	0	0	0	1	1	0	1	00-99	10 YEAR				1 YEAR			
	/W									*POR STATE	0	1	1	1	0	0	0	0
CONTROL	RD	0	0	0	0	1	1	1	1		WP	0	0	0	0	0	0	0
	/W									*POR STATE	0	0	0	0	0	0	0	0
CENTURY	RD	0	0	1	0	0	1	1		00-99	1000 YEAR				100 YEAR			
	/W									*POR STATE	0	0	0	1	1	0	0	1

Note: ***POR STATE** defines power-on reset state of register contents.

Рис. 6.

Текст програми представлено далі.

```
#define F_CPU 8000000

#define LOW 0
#define HIGH 1

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

class Port
{
volatile unsigned char * m_pPort;
public:
Port(volatile unsigned char * pPort, bool isOutput);
void Write(char data);
char Read();
void SetPinOut(char pinNum);
void SetPinIn(char pinNum);
```

```
void SetPin(char pinNum);
void ResetPin(char pinNum);
char GetPinState(char pinName);
};

Port::Port(volatile unsigned char * pPort, bool isOutput)
{
m_pPort = pPort;
if(isOutput)
{
       *(m_pPort-1) = 0xFF;
}
}

void Port::Write(char data)
{
*m_pPort = data;
}

char Port::Read()
{
char value;
value = *(m_pPort - 2);
return value;
}

void Port::SetPinOut(char pinNum)
{
*(m_pPort-1) |= (1 << pinNum);
}

void Port::SetPinIn(char pinNum)
{
*(m_pPort-1) &= ~(1 << pinNum);
}

void Port::SetPin(char pinNum)
{
*m_pPort |= (1 << pinNum);
}

void Port::ResetPin(char pinNum)
{
*m_pPort &= ~(1 << pinNum);
}

char Port::GetPinState(char pinName)
{
char state = 0;
state = (*(m_pPort - 2)) & (1 << pinName);
if(state != 0) return LOW;
else return HIGH;
}

class MySPI
{
public:
MySPI(bool isMaster);
char Send(char data);
char Receive(char data = 0);
};

MySPI::MySPI(bool isMaster)
```

```
{
if(isMaster)
{
	SPCR = 0b01011111;
}
else
{
	SPCR = 0b01001111;
}
}

char MySPI::Send(char data)
{
char received_data = 0;
SPDR = data;
while((SPSR & (1 << SPIF)) == 0);
received_data = SPDR;
return received_data;
}

char MySPI::Receive(char data)
{
char received_data;
while((SPSR & (1 << SPIF)) == 0);
received_data = SPDR;
SPDR = data;
return received_data;
}

class MyUART
{
public:
MyUART();
void SendString(char * str);
};

MyUART::MyUART()
{
UCSRA = 0;
UCSRB = 0b00011000;
UCSRC = 0b00000110;
UBRRH = 0;
UBRRL = 51;
}

void MyUART::SendString(char * str)
{
for(unsigned int i=0;i<strlen(str);i++)
{
	UDR = str[i];
	while((UCSRA & (1<<UDRE)) == 0);
}
}

class RTC_MAX6902
{
MyUART * m_pUart;
Port * m_pPort;
MySPI * m_pSPI;
char m_data[7];
public:
RTC_MAX6902(Port * pPort, MySPI * pSPI, MyUART *pUart);
void GetTime();
```

```
void ShowTime();
};

RTC_MAX6902::RTC_MAX6902(Port * pPort, MySPI * pSPI, MyUART *pUart)
{
m_pPort = pPort;
m_pSPI = pSPI;
m_pUart = pUart;

m_pPort->SetPinOut(4);
m_pPort->SetPinOut(5);
m_pPort->SetPinOut(7);
}

void RTC_MAX6902::GetTime()
{
m_pPort->ResetPin(4);
m_pSPI->Send(0xBF);
m_data[0] = m_pSPI->Send(0); // sec
m_data[1] = m_pSPI->Send(0); //min
m_data[2] = m_pSPI->Send(0); // hour
m_data[3] = m_pSPI->Send(0); //date
m_data[4] = m_pSPI->Send(0); //month
m_data[5] = m_pSPI->Send(0);  // day of week
m_data[6] = m_pSPI->Send(0); // year
m_pPort->SetPin(4);
}

void RTC_MAX6902::ShowTime()
{
char strDateTime[32];
const char days_of_week[7][4] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
GetTime();
sprintf(strDateTime, "%02x:%02x:%02x %02x/%02x/%02x %s\r\n", m_data[2], m_data[1], m_data[0],
m_data[3], m_data[4], m_data[6], days_of_week[(int)(m_data[5] - 1)]);
m_pUart->SendString(strDateTime);
}


int main(void)
{
Port spi_port(&PORTB, false);
MySPI spi_master(true);
MyUART uart;
RTC_MAX6902 clock(&spi_port, &spi_master, &uart);

while (1)
    {
       clock.ShowTime();
       _delay_ms(1000);
    }
}
```

Після запуску системи на моделювання на екрані (рис. 7) з’явиться вікно терміналу та вікно, що відображає значення часу, встановленого в мікросхемі. При моделюванні це значення відповідає системним налаштуванням. Отже, у вікні терміналу буде відображатися той самий час та дата, що і в годиннику.

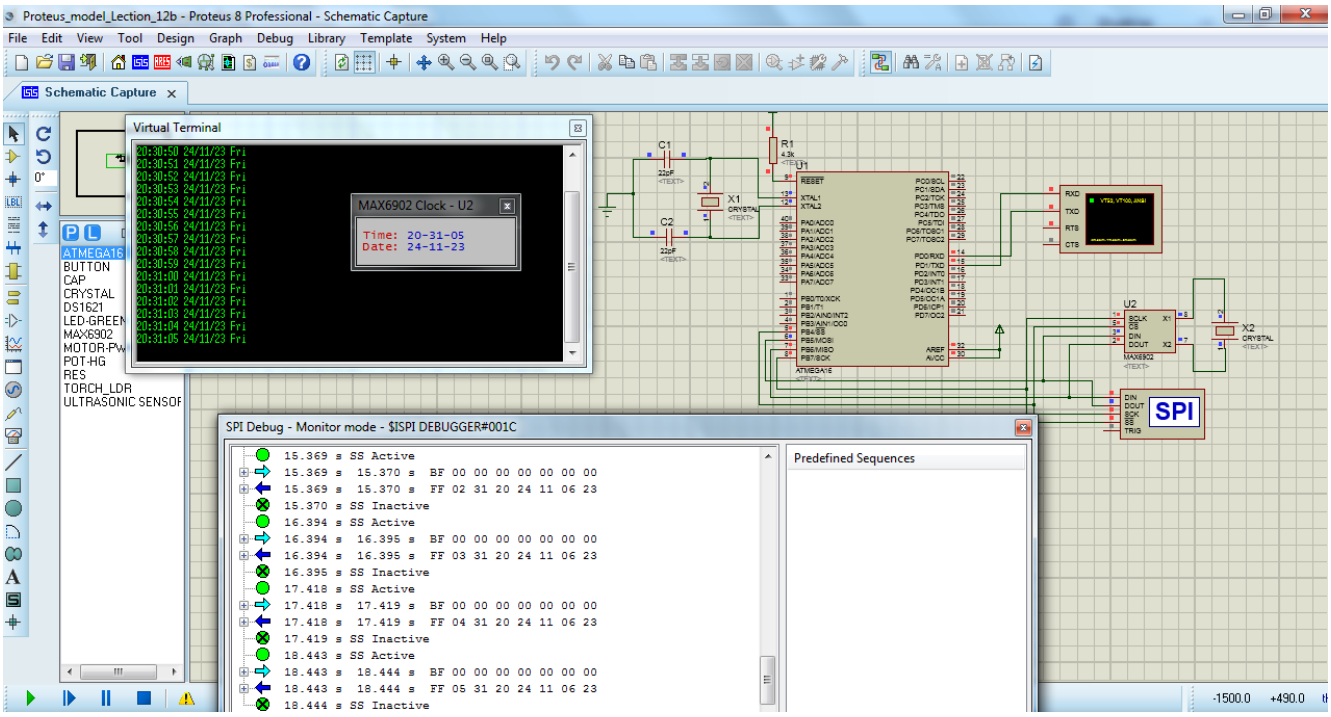


Рис. 7.

Завдання для самостійного виконання:

1. Додати в принципову схему мультимікропроцесорної системи ще один мікроконтролер, який також буде веденим. Підключити До ведучого мікроконтролера ще 4 дискретні кнопки до виводів PC0-PC3 порту C. До другого веденого мікроконтролера до виводів PD0-PD3 порта D підключити чотири світлодіоди (по аналогії з першим мікроконтролером). Схема моделі системи, що розглянута в прикладі, розміщено на Google-диску (файл Proteus_model_Lection_12a.pdsprj).

2. Внести зміни в програму, що розглянута в прикладі, щоб система працювала наступним чином: стан кнопок, що підключені до порту B ведучого мікроконтролера відображається на світлодіодах, що підключені до порту C першого веденого мікроконтролера; стан кнопок, що підключені до порту C ведучого мікроконтролера відображається на світлодіодах, що підключені до порту D другого веденого мікроконтролера. При необхідності внести зміни в протокол обміну даними між мікроконтролерами (це буде залежати від схемотехнічного рішення, яке ви приймете при додаванні в схему третього мікроконтролера, а саме яким чином буде використано інтерфейс SPI для обміну даними між мікроконтролерами – одним ведучим та двома відомими).

3. Оформити звіт, в якому навести розроблену програму. Завантажити звіт на гугл-диск.

Практична робота 13.

Тема: Вивчення принципів роботи з комунікаційним інтерфейсом 1-Wire. Організація процесу зчитування даних з цифрового датчика температури DS18B20 по однопровідному інтерфейсу. Розробка класів, що реалізують високорівневі інтерфейси для взаємодії з апаратними засобами мікроконтролера, для роботи з інтерфейсом 1-Wire та датчиком DS18B20.

Однопровідна шина оперує з TTL-рівнями, тобто логічна одиниця представлена рівнем напруги близько 5 В, а логічний нуль – напругою поблизу 0 В. У вихідному стані на лінії присутній рівень логічної одиниці, що забезпечується резистором номіналом близько 5 кОм.

Ініціатором обміну по однопровідній шині завжди виступає майстер. Всі пересилання починаються із процесу ініціалізації. Ініціалізація виконується в наступній послідовності (рис. 1):

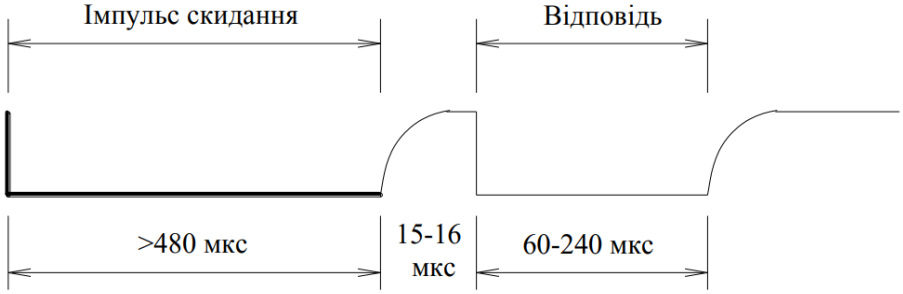


Рис. 1.

Майстер посилає імпульс скидання (reset pulse) – сигнал низького рівня тривалістю не менше 480 мкс. За імпульсом скидання відбувається відповідь підлеглого пристрою (presence pulse) – сигнал низького рівня тривалістю 60 – 240 мкс, що генерується через 15 – 60 мкс після завершення імпульсу скидання.
Відповідь підлеглого пристрою для головного пристрою означає, що на шині присутній термометр і він готовий до обміну даними. Після того, як майстер виявив відповідь, він може передати термометру одну з команд. Передача ведеться шляхом формування майстром спеціальних часових інтервалів (time slots). Кожен часовий інтервал служить для передачі одного біта. Першим передається молодший біт. Інтервал починається імпульсом низького рівня, тривалість якого лежить у межах 1 – 15 мкс. Оскільки перехід з одиниці в нуль менш чутливий до ємності шини (він формується відкритим транзистором, у той час як перехід з нуля в одиницю формується резистором), саме цей перехід використовують однопровідні пристрої для синхронізації з майстром. У підлеглому пристрої запускається схема часової затримки, що визначає момент зчитування даних. Номінальне значення затримки дорівнює 30 мкс, однак воно може коливатися в межах 15 – 60 мкс. За імпульсом низького рівня видається черговий біт для передачі. Він повинен утримуватися майстром на шині протягом 60 – 120 мкс від початку інтервалу. Часовий інтервал завершується переводом шини в стан високого рівня на час не менший 1 мкс. Слід відзначити, що

обмеження на цей час не накладається. Аналогічним чином формуються часові інтервали для всіх інших біт (рис. 2):

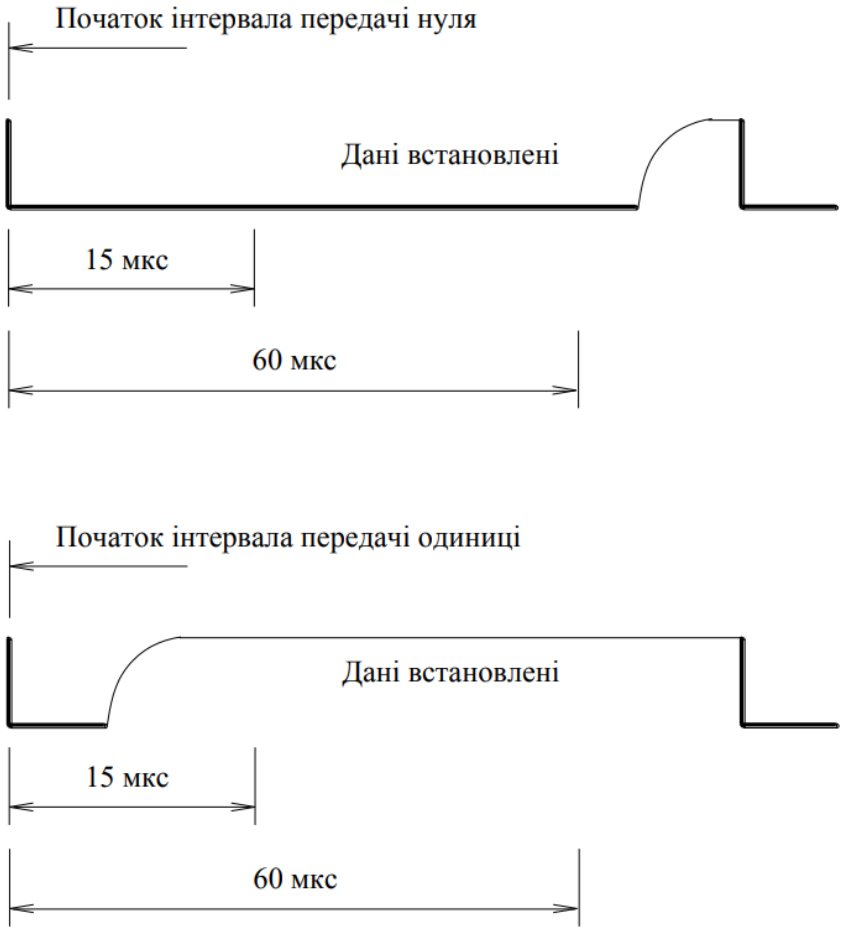


Рис. 2.

Першою командою, яку повинен передати майстер для DS18B20 після ініціалізації, є одна з команд функцій ПЗП. Усього DS18B20 має 5 команд функцій ПЗП:

- Read ROM [33h]. Ця команда дозволяє прочитати вміст ПЗП. У відповідь на цю команду DS18B20 передає 8-бітний код сімейства (10h), потім 48-бітний серійний номер, а потім 8-бітне значення CRC для перевірки правильності прийнятої інформації.
- Match ROM [55h]. Ця команда дозволяє адресувати на шині конкретний термометр. Після цієї команди майстер повинен передати потрібний 64-бітний код, і тільки той термометр, що має такий код, буде «відгукуватися» до наступного імпульсу скидання.
- Skip ROM [CCh]. Ця команда дозволяє пропустити процедуру порівняння серійного номера, і тим самим заощадити час у системах, де на шині є всього один пристрій.
- Search ROM [F0h]. Ця досить складна у використанні команда дозволяє визначити серійні номери всіх термометрів, що присутні на шині.
- Alarm Search [ECh]. Ця команда аналогічна попередній, але «відгукуватися» будуть тільки ті термометри, у яких результат

останнього виміру температури виходить за встановлені межі TH та TL.

Оскільки в нашому прикладі всього один пристрій, найбільш підходящою є функція Skip ROM. Крім неї ще може бути корисною функція Read ROM, що дозволяє ідентифікувати підключений на шину пристрій по його коду сімейства та серійному номеру.

При прийомі даних від підлеглого пристрою часові інтервали для прийнятих біт також формує майстер. Інтервал починається імпульсом низького рівня тривалістю 1 – 15 мкс. Потім майстер повинен звільнити шину, щоб дати можливість термометру вивести біт даних. По переходу з одиниці в нуль DS18B20 виводить на шину біт даних і запускає схему часової затримки, що визначає, як довго біт даних буде присутній на шині. Цей час лежить у межах 15 – 60 мкс. Для того щоб дані на шині, яка завжди має деяку ємність, гарантовано встановилися, потрібний деякий час. Тому момент зчитування даних майстром повинен відстояти якнайдалі, але не більше ніж на 15 мкс від початку часового інтервалу (рис. 3):

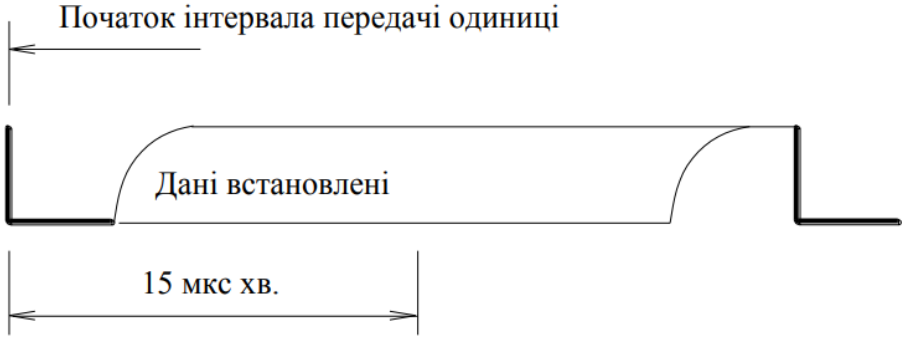


Рис. 3.

Прийом байта починається з молодшого біта. Спочатку передається байт коду сімейства. За кодом сімейства передається 6 байт серійного номера, починаючи з молодшого. Потім пердається байт контрольної суми (CRC). В обчисленні байта контрольної суми беруть участь перші 7 байт, або 56 переданих біт. Після прийому даних майстер повинен обчислити контрольну суму та порівняти значення з CRC. Якщо ці значення збігаються – прийом даних пройшов без помилок.

Після обробки однієї з команд функцій ПЗП, DS18B20 здатний сприймати ще кілька команд:

- Write Scratchpad [4Eh]. Ця команда дозволяє записати дані в проміжний ОЗП DS18B20.
- Read Scratchpad [BEh]. Ця команда дозволяє зчитувати дані із проміжного ОЗП.
- Copy Scratchpad [48h]. Ця команда копіює байти TH і TL із проміжного ОЗП в енергонезалежну пам'ять. Ця операція вимагає близько 10 мс.
- Convert T [44h]. Ця команда запускає процес перетворення температури.

– Recall E2 [B8h]. Ця команда діє зворотним чином відносно команди Copy Scratchpad, тобто вона дозволяє зчитувати байти TH і TL з енергонезалежної пам'яті в проміжний ОЗП. При включенні живлення ця команда виконується автоматично.
– Read Power Supply [B4h]. Ця команда дозволяє перевірити, чи використає DS18B20 паразитне живлення. Справа в тому, що DS18B20 можна підключати всього за допомогою двох проводів. У цьому випадку для живлення використовується лінія даних.

При використанні DS18B20 тільки для виміру температури потрібні всього дві із цих команд: Convert T і Read Scratchpad. Послідовність дій при вимірі температури повинна бути наступною:

1. Посилаємо імпульс скидання і приймаємо відповідь термометра.
2. Посилаємо команду Skip ROM [CCh].
3. Посилаємо команду Convert T [44h].
4. Формуємо затримку мінімум 750 мс.
5. Посилаємо імпульс скидання і приймаємо відповідь термометра.
6. Посилаємо команду Skip ROM [CCh].
7. Посилаємо команду Read Scratchpad [BEh].
8. Читаємо дані із проміжного ОЗП (8 байт) і CRC.
9. Перевіряємо CRC, і якщо дані зчитані вірно, обчислюємо температуру.

Приклад. Виконується розробка програми, що зчитує дані про температуру з цифрового датчика DS18B20, що підключений по інтерфейсу 1-Wire до 6 виводу порту D мікроконтролера. Зчитані з датчика дані про температуру перетворюються у текстовий рядок та виводяться на екрані віртуального терміналу, підключеного по інтерфейсу UART. В програмі створено 4 класи: клас для роботи з портом вводу/виводу, клас для роботи з інтерфейсом 1-Wire, клас для роботи з датчиком DS18B20 та клас для роботи з UART. В кожному з класів реалізовано необхідний інтерфейс (відкриті функції-члени класів) для виконання поставлених завдань.

На рис. 4 представлено принципову схему системи вимірювання температури.

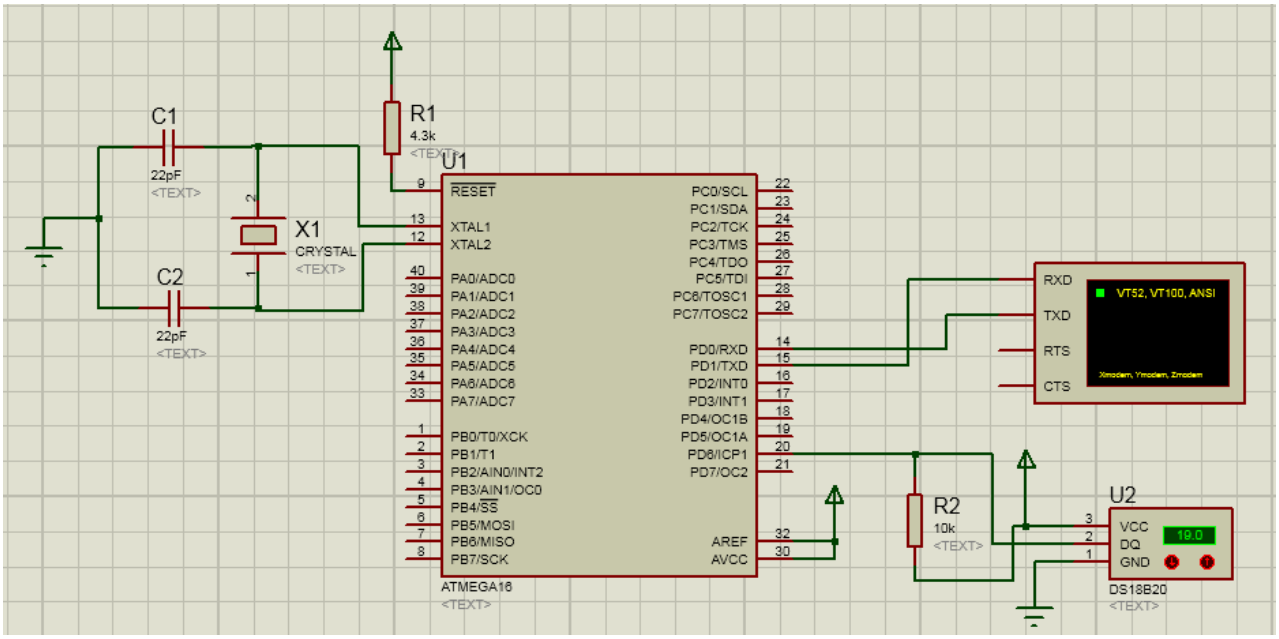


Рис. 4.

Структура програмного забезпечення представлена на рис. 5.

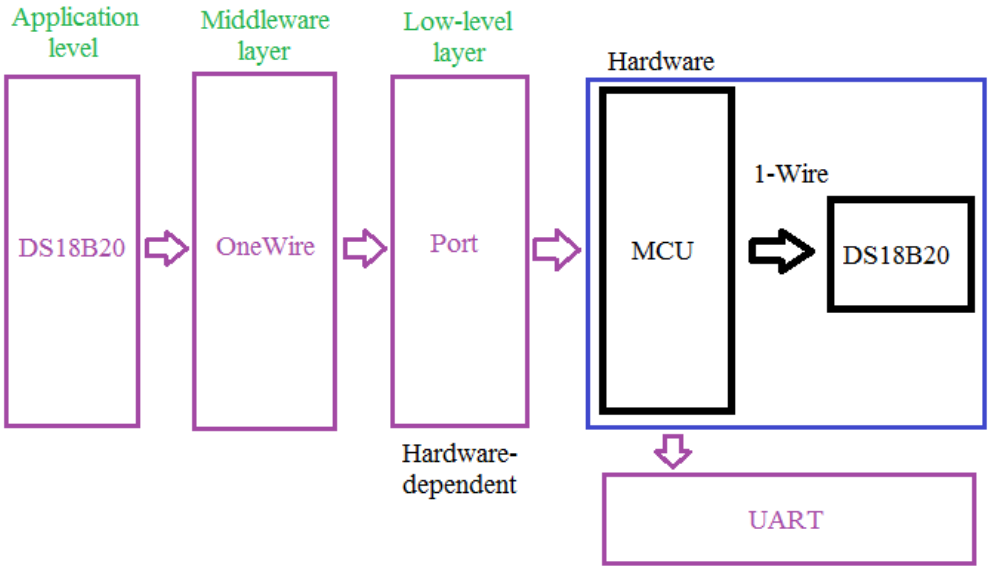


Рис. 5.

Текст програми наведено нижче.

```
#define F_CPU 8000000

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

#define LOW 0
#define HIGH 1
#define WIRE_PORT PORTD
#define PIN_NUM 6
```

```
class Port
{
private:
volatile unsigned char * m_pPort;
public:
Port(volatile unsigned char * pPort);
void SetPinOut(char pinNum);
void SetPinIn(char pinNum);
void SetPinHigh(char pinNum);
void SetPinLow(char pinNum);
char GetPinState(char pinNum);
};

Port::Port(volatile unsigned char * pPort)
{
m_pPort = pPort;
}

void Port::SetPinOut(char pinNum)
{
*(m_pPort-1) |= (1 << pinNum);
}

void Port::SetPinIn(char pinNum)
{
*(m_pPort-1) &= ~(1 << pinNum);
}

void Port::SetPinHigh(char pinNum)
{
*m_pPort |= (1 << pinNum);
}

void Port::SetPinLow(char pinNum)
{
*m_pPort &= ~(1 << pinNum);
}

char Port::GetPinState(char pinNum)
{
if(((*(m_pPort-2)) & (1 << pinNum)) == 0)
{
	return LOW;
}
return HIGH;
}

class OneWire
{
Port * m_pPort;
char m_pinNum;
public:
OneWire(Port * pPort, char pinNum);
char reset();
```

```
void send_bit(char bit);
char receive_bit();
void send_byte(char data);
char read_byte();
};

OneWire::OneWire(Port * pPort, char pinNum)
{
m_pPort = pPort;
m_pinNum = pinNum;
}

char OneWire::reset()
{
char res;
m_pPort->SetPinOut(m_pinNum); // set direction out
_delay_us(245);                          // waiting 480 us
_delay_us(245);

m_pPort->SetPinIn(m_pinNum);// set direction in
_delay_us(65); // waiting answer from device
if(m_pPort->GetPinState(m_pinNum) == LOW)
{
        res = 1;
}
else
{
       res=0;
}
_delay_us(245);                          // waiting 480 us = 60+240+180
_delay_us(185);
return res;
}

void OneWire::send_bit(char bit)
{
m_pPort->SetPinOut(m_pinNum);
_delay_us(1);
if(bit==0)
{
       _delay_us(90); // if is sending 0, pin in output state
}
else
{
       m_pPort->SetPinIn(m_pinNum); // release line for send 1
       _delay_us(90);
}
m_pPort->SetPinIn(m_pinNum); // release line after bit has been sent
_delay_us(1);
}


char OneWire::receive_bit()
{
char state;
```

```
m_pPort->SetPinOut(m_pinNum);
_delay_us(1);
m_pPort->SetPinIn(m_pinNum); // set direction in
_delay_us(14);
if(m_pPort->GetPinState(m_pinNum) == LOW)
{
	state = 0;
}
else
{
	state = 1;
}
_delay_us(50);
return state;
}


void OneWire::send_byte(char data)
{
char i;
for(i=0;i<8;i++)
{
	if((data&0x01)==0) send_bit(0);
	else send_bit(1);
	data=(data>>1);
}
}


char OneWire::read_byte()
{
char i;
char data=0;
for(i=0;i<8;i++)
{
	if(receive_bit()==1) data|=(1<<i);
}
return data;
}


class DS18B20
{
private:
OneWire * m_pOneWire;
public:
DS18B20(OneWire * pOneWire);
int GetTemperature();
};

DS18B20::DS18B20(OneWire * pOneWire)
{
m_pOneWire = pOneWire;
}
```

```
int DS18B20::GetTemperature()
{
int temp;
char data[9];
m_pOneWire->reset();
m_pOneWire->send_byte(0xCC);
m_pOneWire->send_byte(0x44);
_delay_ms(800);

m_pOneWire->reset();
m_pOneWire->send_byte(0xCC);
m_pOneWire->send_byte(0xBE);
data[0]=m_pOneWire->read_byte();
data[1]=m_pOneWire->read_byte();
temp = data[1];
temp = temp<<8;
temp |= data[0];
temp *= 0.0625;
return temp;
}


class Uart
{
public:
Uart();
void SendTemperature(int temp);
};

Uart::Uart()
{
UCSRA = 0;
UCSRB = 0b00011000;
UCSRC = 0b00000110;
UBRRH = 0;
UBRRL = 51;
}

void Uart::SendTemperature(int temp)
{
char str[16];
sprintf(str, "T=%d C\r\n", temp);
for(unsigned int i = 0; i < strlen(str); i++)
{
       UDR = str[i];
       while((UCSRA & (1 << UDRE)) == 0);
}
}

int main(void)
{
int temp;

Port port(&WIRE_PORT);
OneWire oneWire(&port, PIN_NUM);
```

```
DS18B20 sensor(&oneWire);
Uart uart;

while(1)
{
	temp = sensor.GetTemperature();
	uart.SendTemperature(temp);
}
}
```

Після запуску процесу моделювання у вікні терміналу можна бачити зчитане з датчика DS18B20 значення температури (рис. 6).

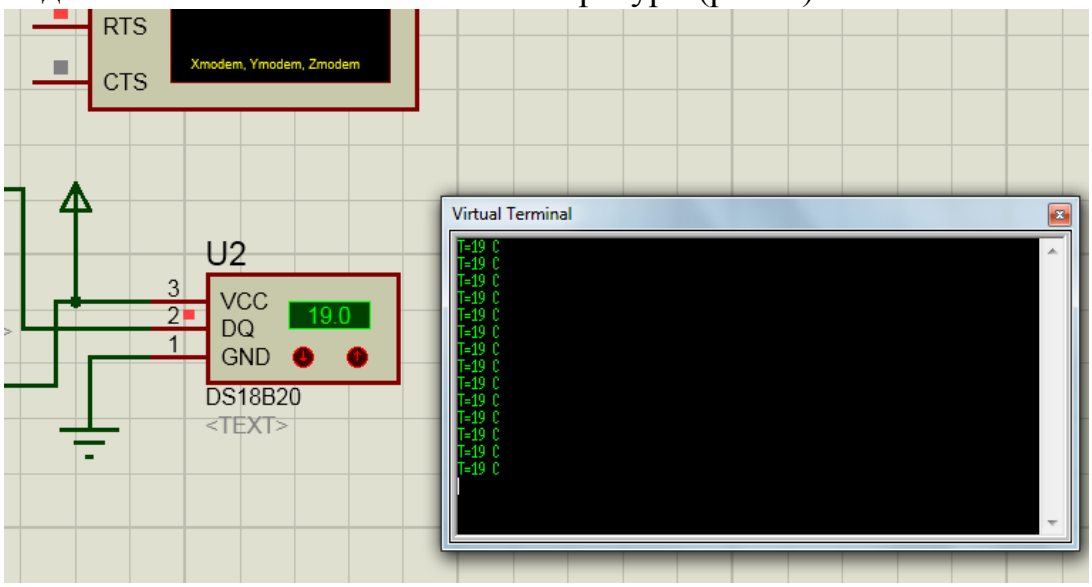


Рис. 6.

Завдання для самостійного виконання:

1. Додати в принципову схему що 2 датчики температури DS18B20 та підключити їх до будь-яких вільних виводів мікроконтролера. Схема моделі системи, що розглянута в прикладі, розміщено на Google-диску (файл Proteus_model_Lection_13.pdsprj).

2. Внести зміни в програму, що розглянута в прикладі, щоб виконувалося зчитування даних з трьох датчиків температури та відображення даних у вікні терміналу. При цьому слід додати додатковий параметр (текстовий рядок, що є масивом символів) в клас датчика температури, який буде задаватися при створенні об’єкту (наприклад, “T1”, “T2”, “T3”), для того, щоб при відображенні значення температури у вікні терміналу було зрозуміло, з якого саме датчика було отримане відповідне значення температури, наприклад так:

T1=21 C

T2=19 C

T3=20 C

3. Додати в схему мікросхему годинника реального часу DS1621, підключеного до мікроконтролера по інтерфейсу I2C. Використовуючи класи для роботи з інтерфейсом I2C та годинником реального часу, що були розроблені на

попередніх заняттях, реалізувати функціонал відображення часу, коли саме було виконано вимірювання температури, та відображати у вікні терміналу разом з виміряним значенням температури.

4. Оформити звіт, в якому навести розроблену програму. Завантажити звіт на гугл-диск.

Практична робота 14.

Тема: Вивчення шаблонів (паттернів) проєктування програмного забезпечення (software design patterns)). Вивчення принципів реалізації та особливостей використання шаблонів Одинак (Singletone), Адаптер (Adapter), Декоратор (Decorator), Спостерігач (Observer), Стратегія (Strategy) для вбудованих мікропроцесорних систем.

Одинак (Singletone).

Сінглтон (Одинак) – це один із найпростіших шаблонів (патернів) проектування, який застосовується до класу. Іноді кажуть: "цей клас – сінглтон", маючи на увазі, що цей клас реалізує патерн проектування сінглтон.

Іноді потрібно написати клас, у якого можна буде створити лише один об'єкт. Наприклад, клас, який відповідає за логування або підключення по одному з комунікаційних інтерфейсів.

Сінглтон (Одинак) – це шаблон (паттерн) проектування, який робить дві речі:

1. Дає гарантію, що у класу буде лише один екземпляр класу.
2. Надає глобальну точку доступу до екземпляра цього класу.

Звідси дві особливості, характерні практично кожної реалізації патерну синглтон:

1. Приватний конструктор – це обмежує можливість створення об'єктів класу поза самим класом.
2. Публічний статичний метод, який повертає екземпляр класу. Цей метод називають Instance(). Це глобальна точка доступу до екземпляра класу.

На рис. 1 представлено UML-діаграму, що описує цей патерн.

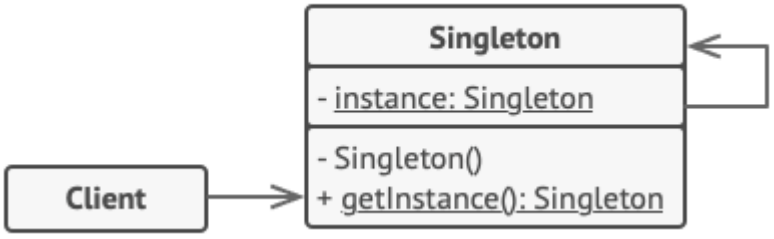


Рис. 1.

В наступному прикладі виконується розробка класу для роботи UART.При розробці цього класу застосовано патерн Одинак. В мікроконтролері, що використовується, наявний лише один UART, тому при спробі створення декількох екземплярів класу для роботи з UART та виклику відповідної функції

інстанціювання, буде створено новий об’єкт, якщо це відбувається вперше, або буде повертатися вказівник на вже існуючий об’єкт, якщо спроба створення цього об’єкта не перша. Тому в програми завжди буде існувати лише один екземпляр класу для роботи з UART, що логічно, адже у мікроконтролера ATMega16 наявний лише один вузол UART.

На рис. 2 представлено схему системи для дослідження роботи програми. В схемі до мікроконтролера підключений термінал, за допомогою якого можна буде спостерігати надіслані по UART дані.

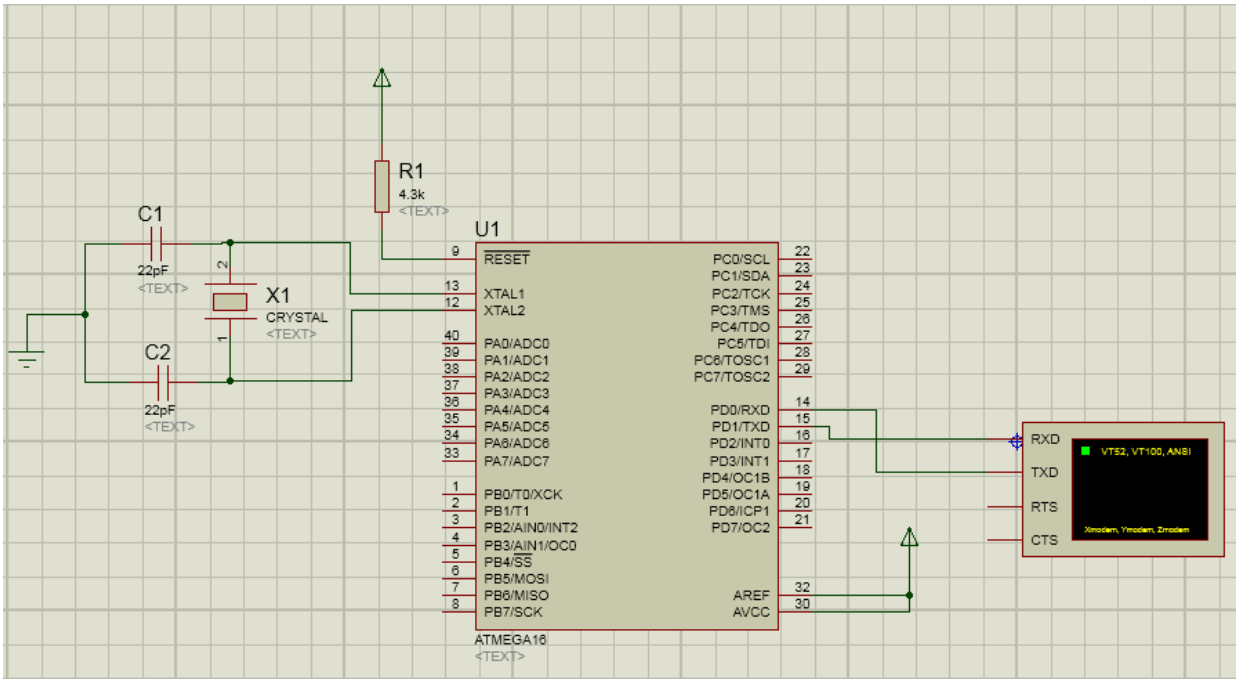


Рис. 2.

Текст програми наведено далі.

```
#define F_CPU 8000000

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

class Uart
{
private:
	static Uart *pUart;
	Uart()
	{
		UCSRA = 0;
		UCSRB = 0b00011000;
		UCSRC = 0b00000110;
		UBRRH = 0;
		UBRRL = 51;
	};

public:
	static Uart* Instance()
	{
		if(pUart == 0)
```

```
		{
			pUart = (Uart*)malloc(sizeof(Uart));
			Uart();
		}
		return pUart;
	}

	void SendString(const char * str)
	{
		for(unsigned int i = 0; i < strlen(str); i++)
		{
			UDR = str[i];
			while((UCSRA&(1<<UDRE)) == 0);
		}
	}

	static void DeleteInstance()
	{
		if(pUart)
		{
			UCSRA = 0;
			UCSRB = 0;
			UCSRC = 0;
			UBRRH = 0;
			UBRRL = 0;
			free(pUart);
			pUart = 0;
		}
	}
};

Uart* Uart::pUart = 0;

void test1()
{
	Uart * p1 = Uart::Instance();
	p1->SendString("Hello from test1()\r\n");
	_delay_ms(1000);
	Uart::DeleteInstance();
}

void test2()
{
	Uart * p2 = Uart::Instance();
	p2->SendString("Hello from test2()\r\n");
	_delay_ms(1000);
	Uart::DeleteInstance();

}

int main(void)
{
    while (1)
    {
		test1();
		test2();
    }
}
```

Результати виконання програми представлені на рис. 3.

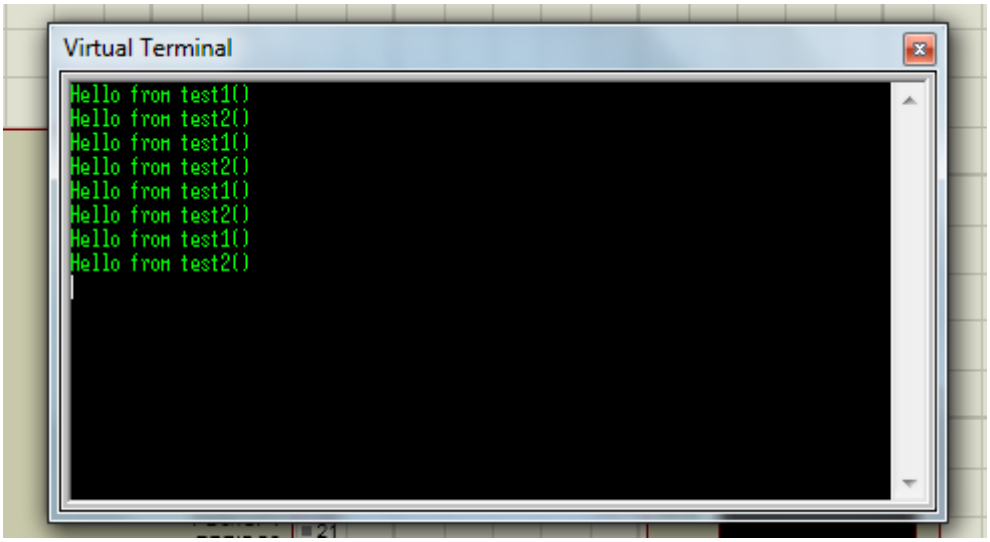


Рис. 3.

Адаптер (Adapter).

Адаптер – це структурний патерн проектування, що дозволяє об'єктам з несумісними інтерфейсами працювати разом. Іншими словами, це об'єкт-перекладач, який трансформує інтерфейс або дані одного об'єкта в такий вигляд, щоб він став зрозумілим іншому об'єкту. При цьому адаптер «обертає» один з об'єктів, тому інший об'єкт навіть не знає про наявність першого. Наприклад, ви можете обернути об'єкт, що працює в метрах, адаптером, який конвертував би дані в фути.

Адаптери можуть не тільки переводити дані з одного формату до іншого, але й допомагати об'єктам з різними інтерфейсами працювати спільно. Це працює так:

1. Адаптер має інтерфейс, який сумісний з одним із об'єктів.
2. Цей об'єкт може викликати методи адаптера.
3. Адаптер отримує ці виклики і перенаправляє їх другого об'єкта, але вже в тому форматі та послідовності, які зрозумілі другому об'єкту.

Реалізація адаптеру, UML-діаграма якої представлена на рис. 4, використовує агрегацію: об'єкт адаптера "обертає", тобто містить посилання на службовий об'єкт. Такий підхід працює у всіх мовах програмування.

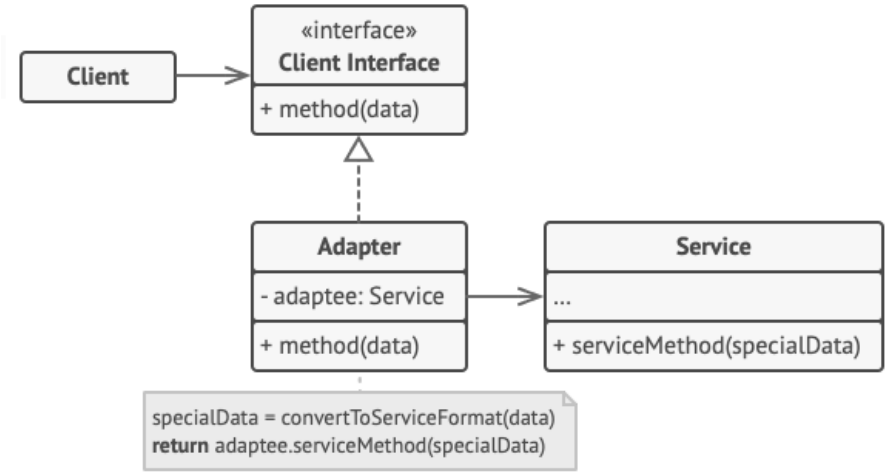


Рис. 4.

На прикладі реалізації системи зчитування температури з аналогового датчика та відправки по UART (рис. 5) демонструється використання патерну Адаптер при розробці програмного забезпечення.

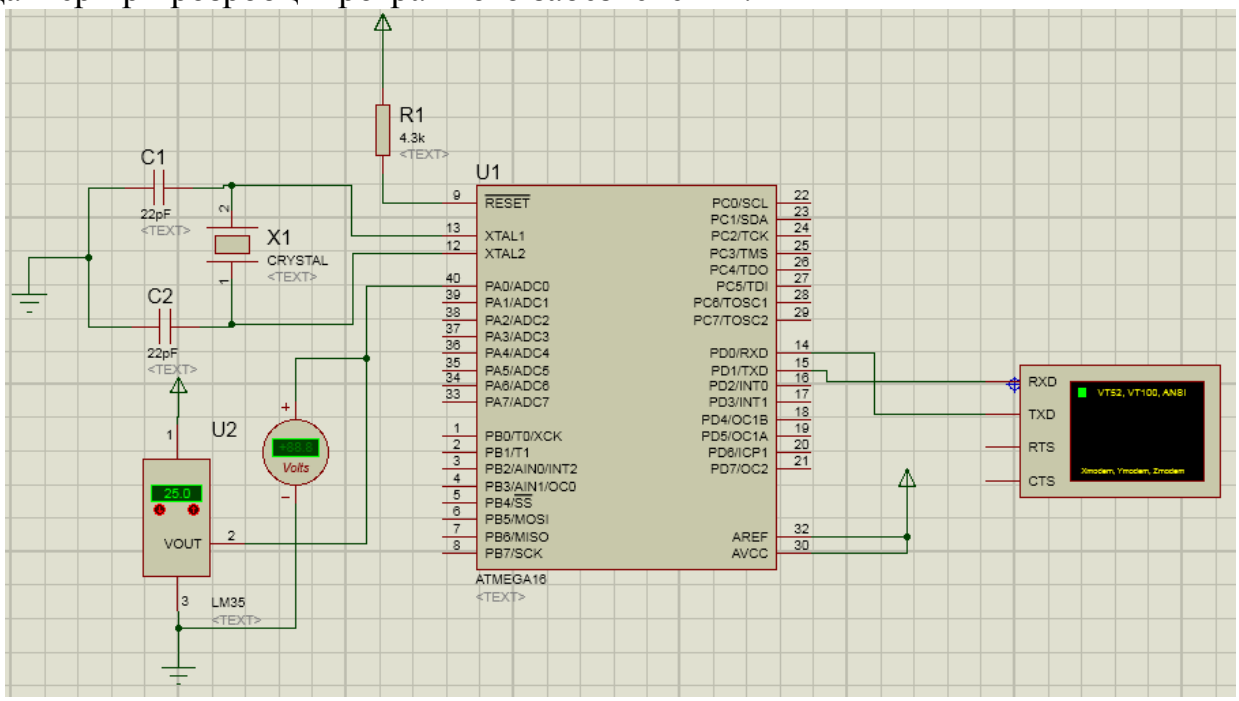


Рис. 5.

Приклад програми без використання патерну Адаптер:

```
#define F_CPU 8000000

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

class Converter
{
public:
	Converter()
	{
		ADCSRA = 0b10000111;
	}

	int GetValue(char channel)
	{
		char dh, dl;
		int result;
		ADMUX = channel;
		ADCSRA |= (1 << ADSC);
		while((ADCSRA & (1 << ADIF)) == 0);
		dl = ADCL;
		dh = ADCH;
		result = dh;
		result = result << 8;
		result = result + dl;
		return result;
	}
};
```

```
class Uart
{
	public:
	Uart()
	{
		UCSRA = 0;
		UCSRB = 0b00011000;
		UCSRC = 0b00000110;
		UBRRH = 0;
		UBRRL = 51;
	}

	void SendString(char * text)
	{
		for(unsigned int i = 0; i < strlen(text); i++)
		{
			UDR = text[i];
			while((UCSRA&(1<<UDRE))==0);
		}
	}
};

int main(void)
{
	char str[16];
	Uart uart;
	Converter adc;

	while(1)
	{
		sprintf(str, "%d\r\n", adc.GetValue(0));
		uart.SendString(str);
		_delay_ms(1000);
	}
}
```

Проблема полягає в перетворенні апаратних одиниць зчитаних з АЦП в фактичне значення температури (в градусах цельсія або фаренгейту) та перетворення значення температури в текстовий рядок для відправки по UART. Для вирішення цього завдання в програмі створюється 2 класи-адаптери. Тобто зручно було б робити так:

```
int main(void)
{
	Uart uart;
	Converter adc;

	while(1)
	{
		uart.SendString(adc.GetValue(0));
		_delay_ms(1000);
	}
}
```

Але інтерфейси об’єктів несумісні, тому при компіляції виникає помилка:

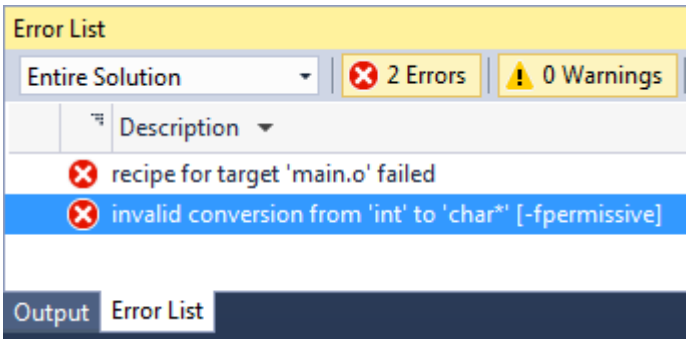


Рис. 6.

Одним зі шляхів подолання цього є використання патерну Адаптер.

```
#define F_CPU 8000000

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

class Converter
{
public:
	Converter()
	{
		ADCSRA = 0b10000111;
	}

	int GetValue(char channel)
	{
		char dh, dl;
		int result;
		ADMUX = channel;
		ADCSRA |= (1 << ADSC);
		while((ADCSRA & (1 << ADIF)) == 0);
		dl = ADCL;
		dh = ADCH;
		result = dh;
		result = result << 8;
		result = result + dl;
		return result;
	}
};

class Uart
{
	public:
	Uart()
	{
		UCSRA = 0;
		UCSRB = 0b00011000;
		UCSRC = 0b00000110;
		UBRRH = 0;
		UBRRL = 51;
	}

	void SendString(char * text)
	{
		for(unsigned int i = 0; i < strlen(text); i++)
		{
			UDR = text[i];
			while((UCSRA&(1<<UDRE))==0);
```

```
                }
        }
};

class ValueToCelsium
{
        char m_sTemp[16];
        Converter * m_pConverter;
public:
        ValueToCelsium(Converter * pConverter)
        {
                m_pConverter = pConverter;
        }

        char* GetValue(char channel)
        {
                int value;
                int temperature;
                value = m_pConverter->GetValue(channel);
                temperature = value * 0.49;
                sprintf(m_sTemp, "T=%d C\r\n", temperature);
                return m_sTemp;
        }
};

class ValueToFarenheit
{
        char m_sTemp[16];
        Converter * m_pConverter;
        public:
        ValueToFarenheit(Converter * pConverter)
        {
                m_pConverter = pConverter;
        }

        char* GetValue(char channel)
        {
                int value;
                int temperature;
                value = m_pConverter->GetValue(channel);
                temperature = (int)(double)(value * 0.49)*9/5 + 32;
                sprintf(m_sTemp, "T=%d F\r\n", temperature);
                return m_sTemp;
        }
};

int main(void)
{
        Uart uart;
        Converter adc;
        ValueToCelsium celsiumAdapter(&adc);
        ValueToFarenheit farenheitAdapter(&adc);

        while(1)
        {
                uart.SendString(celsiumAdapter.GetValue(0));
                uart.SendString(farenheitAdapter.GetValue(0));
                _delay_ms(1000);
        }
}
```

В результаті моделювання роботи системи можна бачити виміряне значення температури в градусах Цельсія та Фаренгейта.

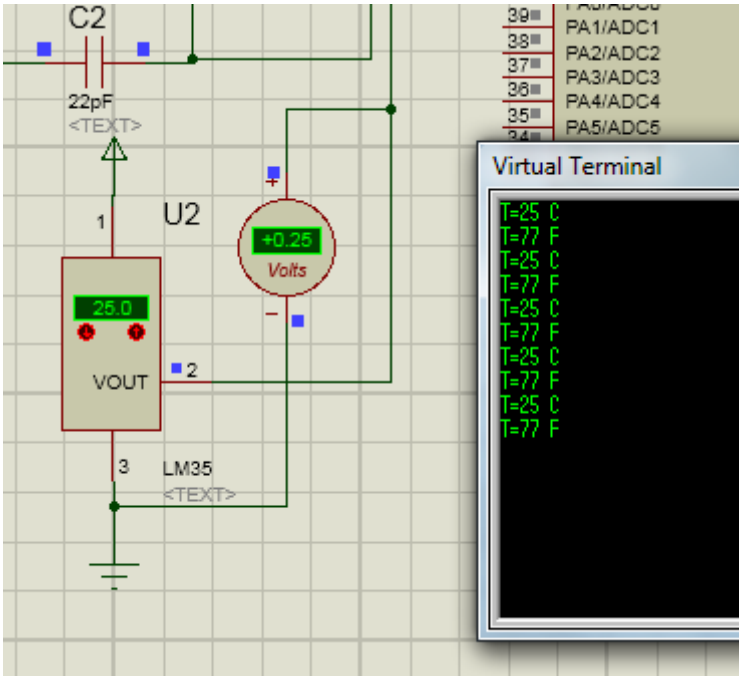


Рис. 7.

Декоратор (Decorator).

Декоратор – це структурний патерн проектування, який дозволяє динамічно додавати об'єктам нову функціональність, обертаючи їх у корисні обгортки.

На рис. 8 представлена UML-діаграма, що пояснює структуру патерну Декоратор. Компонент задає загальний інтерфейс обгорток і об'єктів, що обертаються.Базовий декоратор зберігає посилання вкладений об'єкт-компонент. Їм може бути як конкретний компонент, так і один із конкретних декораторів. Базовий декоратор делегує всі операції вкладеному об'єкту. Додаткова поведінка житиме у конкретних декораторах. Конкретний компонент визначає клас об'єктів, що обертаються. Він містить якусь базову поведінку, яку потім змінюють декоратори.

Конкретні декоратори – це різні варіації декораторів, які містять додаткову поведінку. Воно виконується до або після виклику аналогічної поведінки оберненого об'єкта. Клієнт може обертати прості компоненти та декоратори до інших декораторів, працюючи з усіма об'єктами через загальний інтерфейс компонентів.

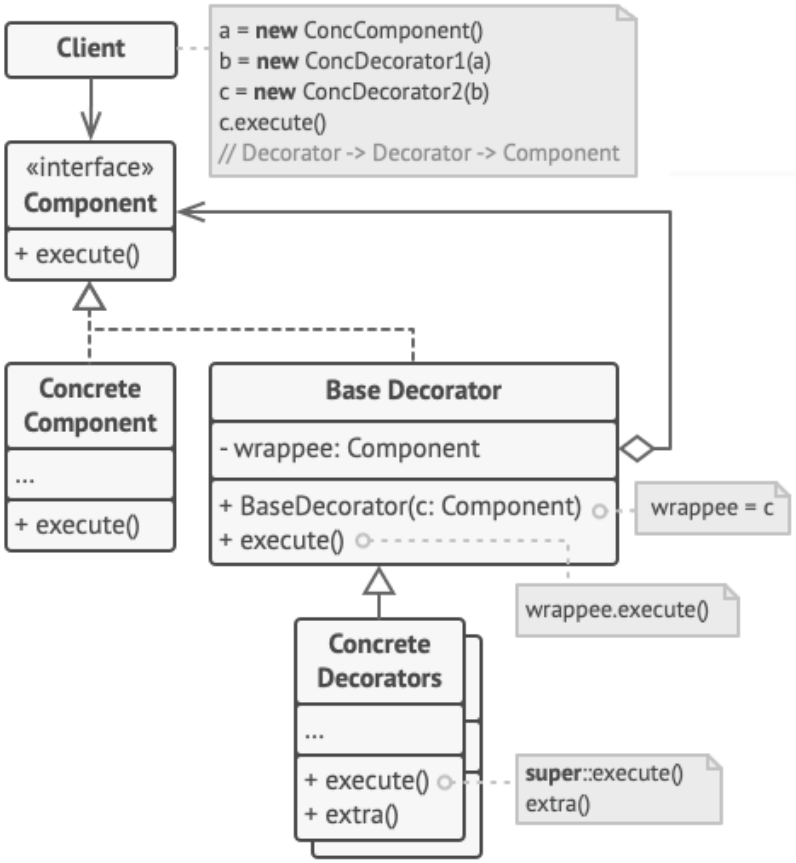


Рис. 8.

Для демонстрації використання цього патерну можна використовувати наступну схему (рис. 9).

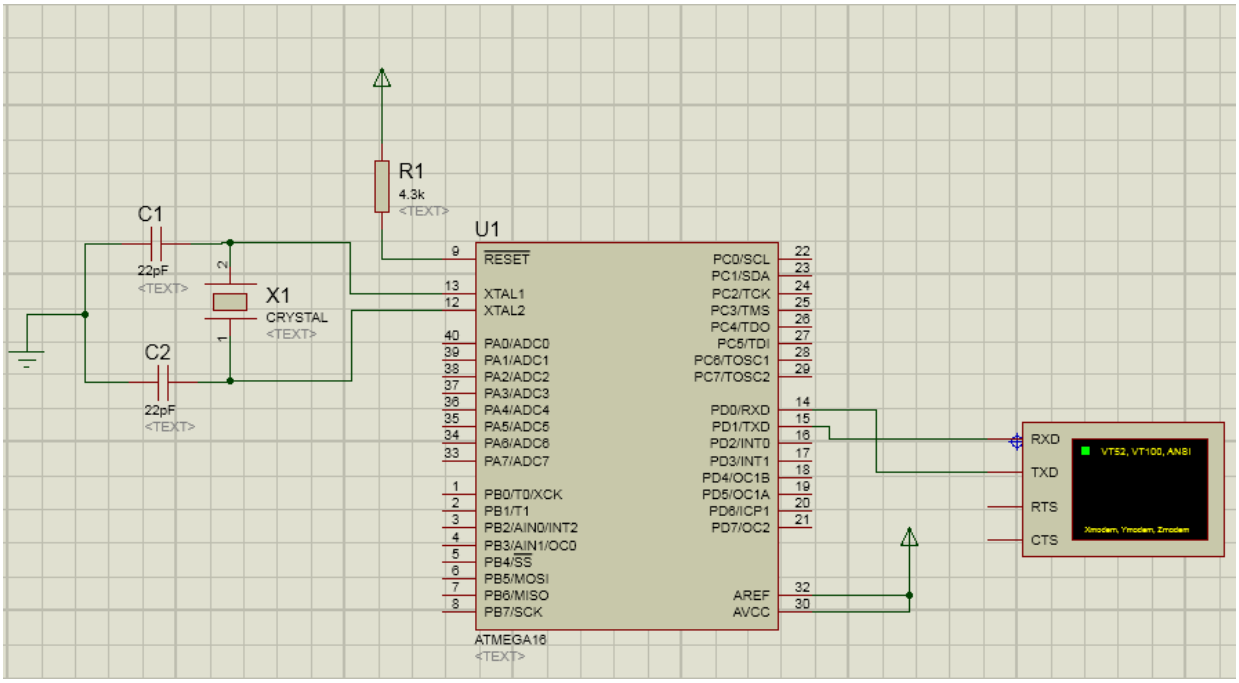


Рис. 9.

В програмі виконується відправка чисел починаючи з 0, з інкрементом 1, які декоруються додатковими позначеннями на початку та в кінці рядка, та

відправляються по UART. При запуску процесу моделювання роботи схеми у вікні терміналу можна буде побачити результат роботи програми.

```
#define F_CPU 8000000

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


class String
{
protected:
	char m_text[64];

public:
	String(char * str)
	{
		strcpy(m_text, str);
	}

	String(int value)
	{
		sprintf(m_text, "%d", value);
	}

	char* GetString()
	{
		return m_text;
	}

	void SetString(char * str)
	{
		strcpy(m_text, str);
	}
};

class StringParameterT : public String
{

public:
	StringParameterT(char * str) : String(str)
	{
	char text_param[64] = "D=";
	strcat(text_param, m_text);
	strcpy(m_text, text_param);
	}

	char * GetString()
	{
		return m_text;
	}
};

class StringUnitsC : public String
{
public:
	StringUnitsC(char * str) : String(str)
	{
		strcat(m_text, " cm");
	}

	char * GetString()
```

```
	{
		return m_text;
	}
};

class StringNewLine : public String
{
public:
	StringNewLine(char * str) : String(str)
	{
		strcat(m_text, "\r\n");
	}

	char * GetString()
	{
		return m_text;
	}
};

class Uart
{
public:
	Uart()
	{
		UCSRA = 0;
		UCSRB = 0b00011000;
		UCSRC = 0b00000110;
		UBRRH = 0;
		UBRRL = 51;
	}

	void SendString(char * text)
	{
		for(unsigned int i = 0; i < strlen(text); i++)
		{
			UDR = text[i];
			while((UCSRA&(1<<UDRE))==0);
		}
	}
};

int main(void)
{
    int value = 0;
	String str(value);
	Uart uart;


    while (1)
    {
	String
decorated_str(StringNewLine(StringUnitsC(StringParameterT(String(value).GetString()).GetString
()).GetString()).GetString());
	uart.SendString(decorated_str.GetString());
	value++;
	_delay_ms(1000);
    }
}
```

Спостерігач (Observer).

Спостерігач – це поведінковий патерн проектування, який створює механізм, що дозволяє одним об'єктам стежити та реагувати на події, що відбуваються в інших об'єктах.

Видавцями називаються ті об'єкти, які містять важливий чи цікавий для інших об'єктів стан. Інші об'єкти, які хочуть відстежувати зміни цього стану, назвемо Передплатниками.

Паттерн Спостерігач пропонує зберігати в об'єкті видавця список посилань на об'єкти передплатників, причому видавець не повинен вести список підписки самостійно. Він надасть методи, за допомогою яких передплатники могли б додавати або прибирати себе зі списку.

Коли у видавця відбуватиметься важлива подія, він проходитиме за списком передплатників і сповіщатиме їх про це, викликаючи певний метод об'єктів-передплатників.

Видавцеві байдуже, який клас матиме той чи інший передплатник, оскільки всі вони повинні дотримуватися загального інтерфейсу та мати єдиний метод оповіщення.

Структура патерна представлена на рис. 10.

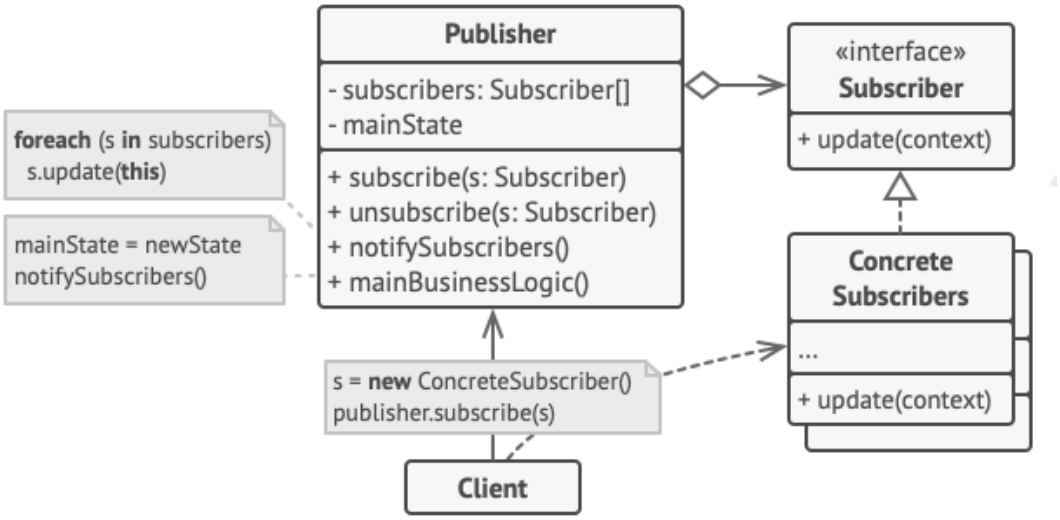


Рис 10.

Для демонстрації використання патерну наведено наступний приклад. При зміні значення напруги на вході АЦП (рис. 11) відбувається відправка текстового рядка по UART з інформацією про зміну значення та перемикання світлодіоду, що підключений до виводу PC0.

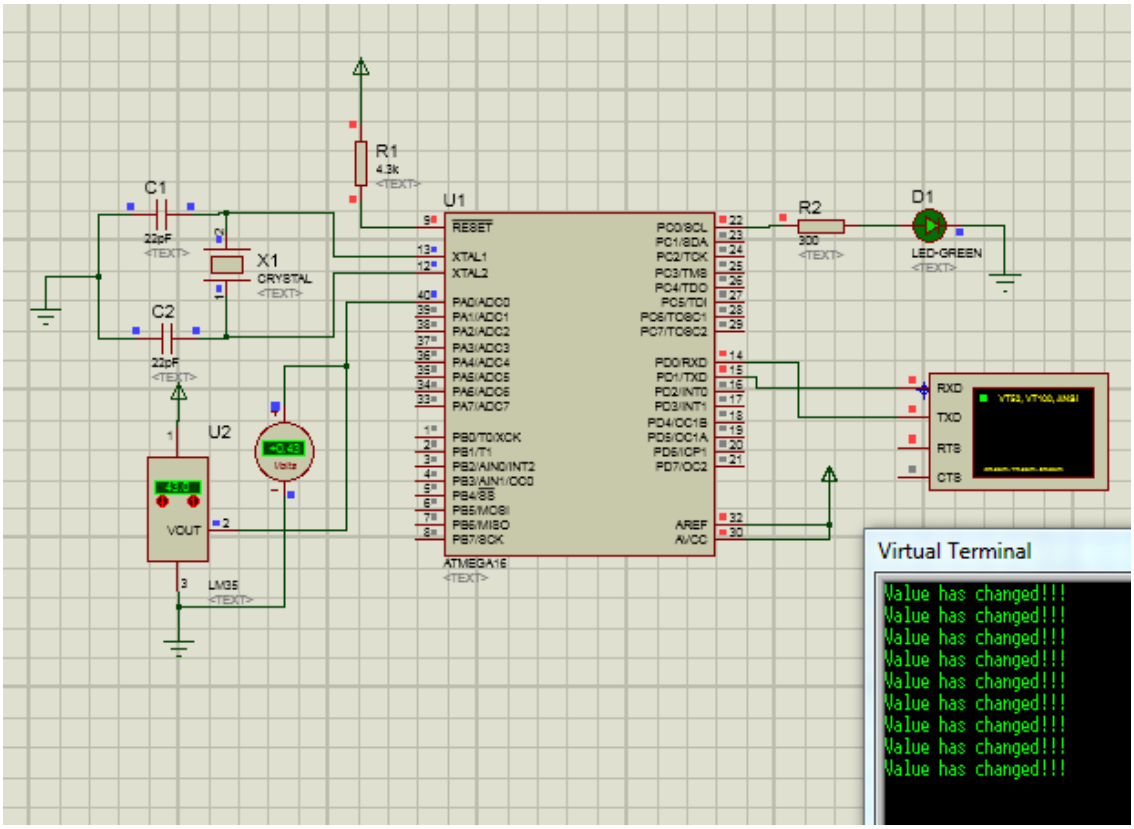


Рис. 11.

Текст програми має вигляд:

```
#define F_CPU 8000000

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

class Converter
{
public:
	Converter()
	{
		ADCSRA = 0b10000111;
	}

	int GetValue(char channel)
	{
		char dh, dl;
		int result;
		ADMUX = channel;
		ADCSRA |= (1 << ADSC);
		while((ADCSRA & (1 << ADIF)) == 0);
		dl = ADCL;
		dh = ADCH;
		result = dh;
		result = result << 8;
		result = result + dl;
		return result;
	}
};

class Uart
```

```
{
	public:
	Uart()
	{
		UCSRA = 0;
		UCSRB = 0b00011000;
		UCSRC = 0b00000110;
		UBRRH = 0;
		UBRRL = 51;
	}

	void SendString(char * text)
	{
		for(unsigned int i = 0; i < strlen(text); i++)
		{
			UDR = text[i];
			while((UCSRA&(1<<UDRE))==0);
		}
	}
};

class Observer
{
public:
	Observer()
	{
		// Empty constructor
	};

	virtual void Notify(int value)
	{

	}
};

class UartObserver : public Observer
{
public:
	UartObserver() : Observer()
	{
		// Nothing to do
	}

	void Notify(int value)
	{
		Uart uart;
		uart.SendString("Value has changed!!!\r\n");
	}
};

class LEDObserver : public Observer
{
	public:
	LEDObserver() : Observer()
	{
		DDRC = 0x01;
	}

	void Notify(int value)
	{
		PORTC ^= 0x01;
	}
};


class ADCSubject
```

```
{
	Converter* m_pConverter;
	int m_currentValue;
	int m_previousValue;
	Observer* m_pObservers[10];
	int m_count;

public:
	ADCSubject(Converter * pConverter)
	{
		m_pConverter = pConverter;
		m_currentValue = m_previousValue = m_pConverter->GetValue(0);
		m_count = 0;
		for(int i=0;i<10;i++)
		{
			m_pObservers[i] = 0;
		}
	}

	bool Subscribe(Observer * pObserver)
	{
		if(m_count < 10)
		{
			m_pObservers[m_count] = pObserver;
			m_count++;
			return true;
		}
		return false;
	}

	bool Unsubscribe(Observer * pObserver)
	{
		for(int i = 0; i < 10; i++)
		{
			if(m_pObservers[i] == pObserver)
			{
				m_pObservers[i] = 0;
				return true;
			}
		}
		return false;
	}

	void CheckADCValue()
	{
		m_currentValue = m_pConverter->GetValue(0);
		if((m_currentValue - m_previousValue) != 0)
		{
			for(int i=0;i<10;i++)
			{
				if(m_pObservers[i] != 0)
				{
					m_pObservers[i]->Notify(m_currentValue);
				}
			}
		}
		m_previousValue = m_currentValue;
	}
};


int main(void)
{
	Converter adc;
	ADCSubject subject(&adc);
	UartObserver uartObs;
```

```
        LEDObserver ledObs;

        subject.Subscribe(&uartObs);
        subject.Subscribe(&ledObs);

        while(1)
        {
                subject.CheckADCValue();
                _delay_ms(500);
        }
}
```

Стратегія (Strategy).

Стратегія – це поведінковий патерн проектування, який визначає сімейство подібних алгоритмів і поміщає кожен з них у свій клас, який називається стратегією, після чого алгоритми можна взаємозамінювати безпосередньо під час виконання програми.

Замість того, щоб початковий клас сам виконував той чи інший алгоритм, він відіграватиме роль контексту, посилаючись на одну із стратегій та делегуючи їй виконання роботи. Щоб змінити алгоритм, достатньо підставити в контекст інший об'єкт-стратегію.

Важливо, щоб усі стратегії мали спільний інтерфейс. Використовуючи цей інтерфейс контекст буде незалежним від конкретних класів стратегій. З іншого боку, буде можливість змінювати та додавати нові види алгоритмів, не чіпаючи код контексту. На рис. 12 представлена структура цього патерну.

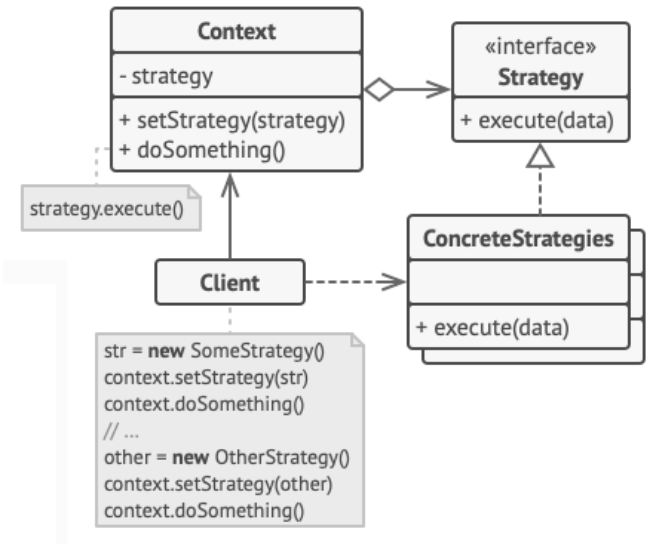


Рис. 12

Для демонстрації реалізації патерну Стратегія використовується схема, представлена на рис. 13. На прикладі створення світлових ефектів «біжучий рядок», «вогні, щозіштовхуються», «блимаючі вогники» на світлодіодах показано, як кожен з алгоритмів реалізації відповідного ефекту інкапсульовано в окремому класі стратегії.

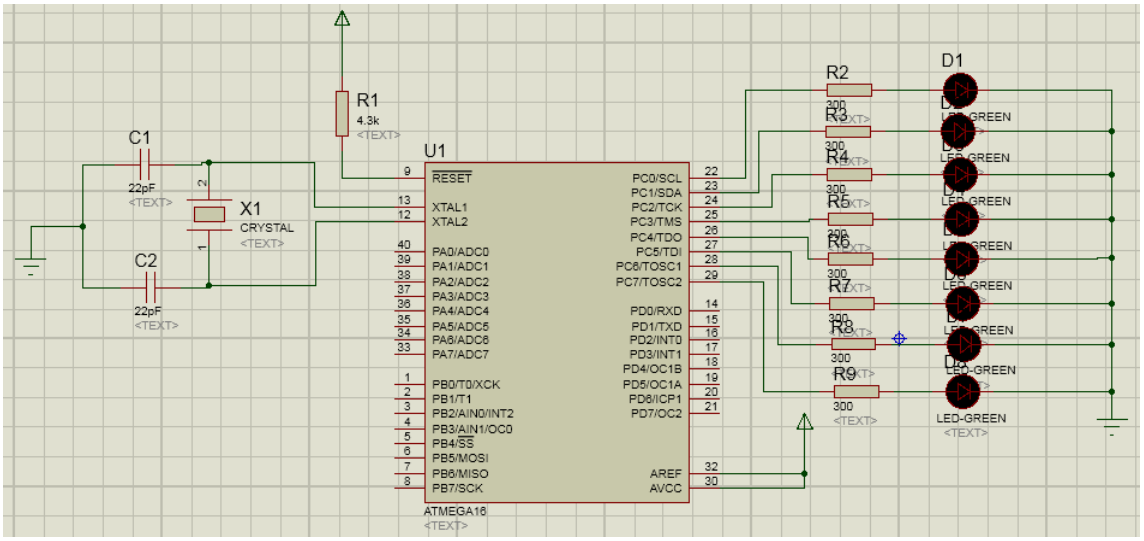


Рис. 13.

Програмна реалізація патерну Стратегія наведена далі.

```
#define F_CPU 8000000

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

class Strategy
{
protected:
	volatile unsigned char * m_pPort;

public:
	Strategy(volatile unsigned char * pPort)
	{
		m_pPort = pPort;
		*(m_pPort-1) = 0xFF; // LEDs connected to PORTC, so set the output mode for this
port
	};

	virtual void Execute()
	{
		// This method will be overwritten in child classes
	}
};

class RunninLights : public Strategy
{
public:
	RunninLights(volatile unsigned char * pPort) : Strategy(pPort)
	{
		// Nothing to do here
	}

	void Execute()
	{
		for(int i=0;i<9;i++)
		{
			*m_pPort = (1 << i);
			_delay_ms(200);
		}
	}
```

```
};

class BlinkingLights : public Strategy
{
	public:
	BlinkingLights(volatile unsigned char * pPort) : Strategy(pPort)
	{
		// Nothing to do here
	}

	void Execute()
	{
		char data[8] = {0x81, 0x42, 0x24, 0x18, 0x24, 0x42, 0x81, 0};
		for(int i = 0; i < 8; i++)
		{
			*m_pPort = data[i];
			_delay_ms(200);
		}
	}
};

class BumpingLights : public Strategy
{
	public:
	BumpingLights(volatile unsigned char * pPort) : Strategy(pPort)
	{
		// Nothing to do here
	}

	void Execute()
	{
		*m_pPort = 0xFF;
		_delay_ms(200);
		*m_pPort = 0;
		_delay_ms(200);
	}
};

class Context
{
private:
	Strategy * m_pStrategy;
public:
	Context()
	{
		m_pStrategy = 0;
	}
	void SetStrategy(Strategy * pStrategy)
	{
		m_pStrategy = pStrategy;
	}

	void ExecuteStrategy()
	{
		m_pStrategy->Execute();
	}
};

int main(void)
{
	Context context;
	RunninLights strategy1(&PORTC);
	BlinkingLights strategy2(&PORTC);
	BumpingLights strategy3(&PORTC);

	context.SetStrategy(&strategy1);
```

```
        while(1)
        {
                context.ExecuteStrategy();
        }
}
```

Отже, патерн Стратегія дозволяє варіювати поведінку об'єкта під час виконання програми, підставляючи в нього різні об'єкти поведінки. Стратегія дозволяє винести поведінку, що відрізняється, в окрему ієрархію класів, а потім звести початкові класи до одного, зробивши поведінку цього класу настроюваним.

Ітератор

Ітератор – це поведінковий патерн проектування, який дає можливість послідовно отримувати доступ до елементів складових об'єктів (масиви, списки та інші колекції), не розкриваючи їх внутрішньої будови.

Колекції – найпоширеніша структура даних, яку можна зустріти у програмуванні. Це набір об'єктів, зібраний в одну купу за якимись критеріями. В контексті вбудованих систем у якості прикладу колекції можна розглядати набір датчиків, наприклад датчиків температури. Для контролю температури у різних точках приміщення потрібно використовувати декілька датчиків температури. Оскільки в такому випадку всі датчики матимуть дещо спільне (всі датчики – це датчики температури, і всі датчики виконують вимірювання температури в різних точках одного і того ж приміщення), тому їх можна об’єднати в деяку структуру даних, наприклад однозв’язний список. Тоді, для зчитування даних з датчиків потрібно пройтись по усіх елементах списку і зчитати виміряне значення температури. І як раз для того, щоб можна було послідовно отримати доступ до кожного елемента списку і використовується ітератор.

Ідея патерна Ітератор полягає в тому, щоб винести поведінку обходу колекції із самої колекції до окремого класу.

Об'єкт-ітератор відстежуватиме стан обходу, поточну позицію в колекції та скільки елементів залишилося обійти. Одну й ту саму колекцію зможуть одночасно використовувати різні ітератори, а сама колекція навіть не знатиме про це. До того ж, якщо потрібно буде додати новий спосіб обходу (наприклад, не послідовний, або не від початку до кінця, а навпаки, з кінця до початку), можна буде створити окремий клас ітератора, не змінюючи існуючий код колекції.

На рис. 14 представлена структура патерну Ітератор.

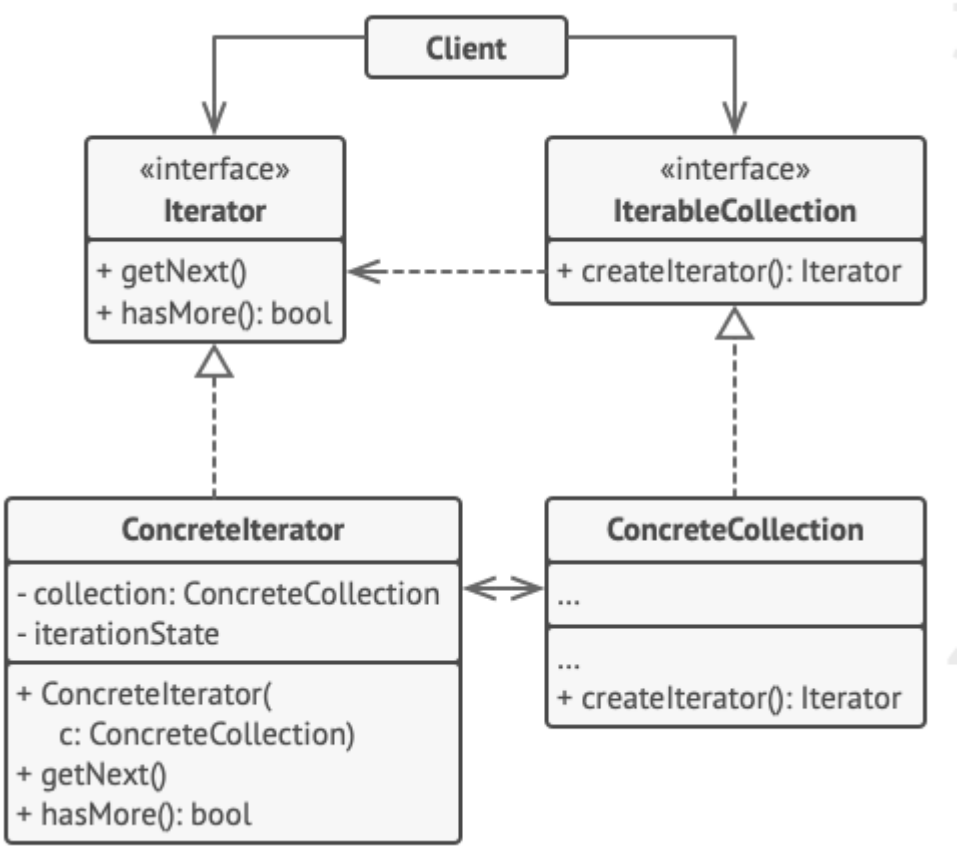


Рис. 14.

Ітератор описує інтерфейс для доступу та обходу елементів колекції. Конкретний ітератор реалізує алгоритм обходу певної колекції. Об'єкт ітератора повинен сам відстежувати поточну позицію при обході колекції, щоб окремі ітератори могли обходити ту саму колекцію незалежно. Колекція визначає інтерфейс отримання ітератора з колекції. Сама колекція може створювати ітератори, оскільки вона знає, які саме ітератори здатні працювати з нею. Конкретна колекція повертає новий екземпляр певного конкретного ітератора, зв'язавши його з об'єктом колекції. Клієнт працює з усіма об'єктами через інтерфейси колекції та ітератора. Так клієнтський код залежить від конкретних класів, що дозволяє застосовувати різні ітератори, не змінюючи існуючий код програми. Зазвичай клієнти не створюють об'єкти ітераторів, а отримують їх із колекцій. Проте, якщо клієнту потрібен спеціальний ітератор, він може створити його самостійно.

В даному прикладі патерн Ітератор використовується для обходу списку датчиків температури. Нв рис. 15 представлена принципова схема системи. До мікроконтролера підключено 4 аналогових датчика температури LM35 (канали 0-3 АЦП).

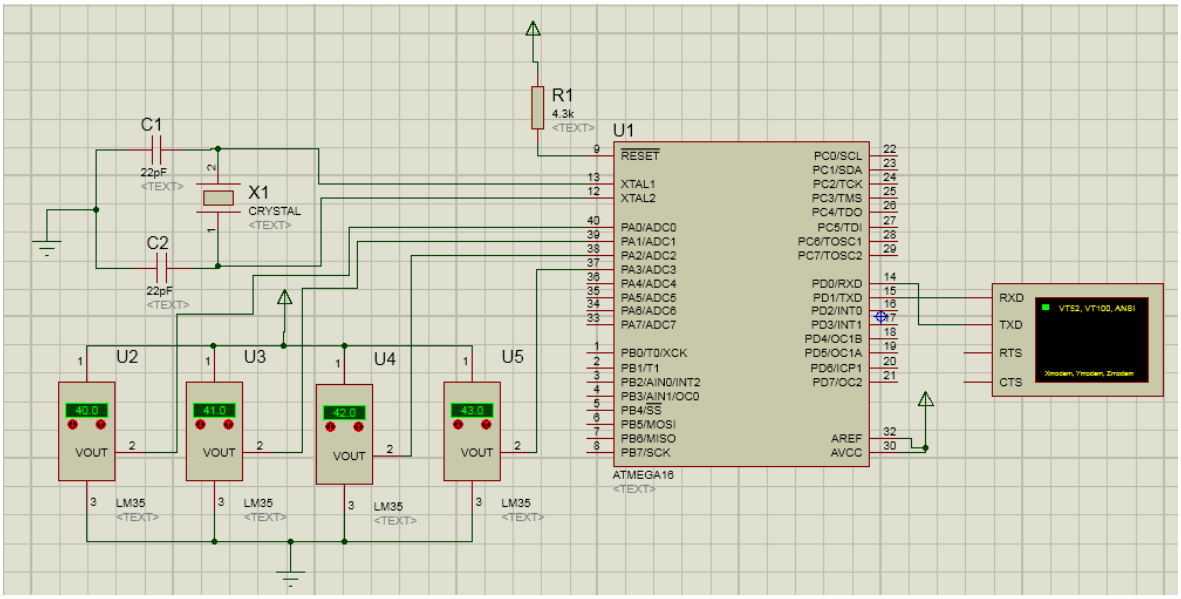


Рис. 15.

Текст програми представлено далі.

```
#define F_CPU 8000000

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

#define CHANNEL_0 0
#define CHANNEL_1 1
#define CHANNEL_2 2
#define CHANNEL_3 3
```

Реалізація класу для роботи з АЦП. В конструкторі класу виконується вмикання АЦП та налаштування його тактової частоти шляхом запису відповідного значення у регістр керування ADCSRA. Клас містить метод, у якому реалізовано запуск процесу аналого-цифрового вимірювання. Цей метод повертає отримане 10-розрядне значення по завершенн процесу перетворення аналогового сигналу.

```
class Converter
{
	public:
	Converter()
	{
		ADCSRA = 0b10000111;
	}

	int GetValue(char channel)
	{
		char dh, dl;
		int result;
		ADMUX = channel;
		ADCSRA |= (1 << ADSC);
		while((ADCSRA & (1 << ADIF)) == 0);
		dl = ADCL;
```

```
            dh = ADCH;
            result = dh;
            result = result << 8;
            result = result + dl;
            return result;
        }
};
```

Наступник клас використовується безпосередньо для роботи з аналгоговим датчиком температури LM35. Фактично цей клас є оболонкою для класу Converter, що використовується для роботи з АЦП. Ці два класи пов’язані відношенням асоціації, тому поле класу LM35_Sensor містить вказівник на клас Converter. Також ще одне поле m_channel використовується для зберігання номра каналу АЦП, до якого підключено датчик температури. Для отримання значення температури (в градусах Цельсія) може бути використаний GetTemperature().

```
class LM35_Sensor
{
        char m_channel;
        Converter * m_pConverter;
public:
        LM35_Sensor(Converter * pConverter, char channel)
        {
            m_pConverter = pConverter;
            m_channel = channel;
        }

        int GetTemperature()
        {
            int value;
            value = m_pConverter->GetValue(m_channel) * 0.49;
            return value;
        }
};
```

Цей клас використовується як драйвер універсального асинхронного пріймача/передавача і дозволяє відправляти символи (або рядки) по UART. Об’єкт цього класу буде використано в прикладі для того, щоб відобразити у вікні терміналу виміряні значення температури, отримані з різних датчиків.

```
class Uart
{
        public:
        Uart()
        {
            UCSRA = 0;
            UCSRB = 0b00011000;
            UCSRC = 0b00000110;
            UBRRH = 0;
            UBRRL = 51;
        }

        void SendString(char * text)
        {
            for(unsigned int i = 0; i < strlen(text); i++)
            {
                UDR = text[i];
                while((UCSRA&(1<<UDRE))==0);
            }
        }
};
```

Цей клас описує елемент списку. Однозв'язний список – це структура даних, що складається з вузлів. Кожен вузол буде мати якесь значення (в даному прикладі вказівник на об’єкт датчика температури) і вказівник на наступний вузол. У цьому вузлі є:

– вказівник на об”єкт датчика температури;
– вказівник на наступний елемент (за замовчуванням null);
– конструктор.

```
class Item
{
public:
	Item * m_pNext;
	LM35_Sensor * m_pSensor;
	Item(LM35_Sensor * pSensor)
	{
		m_pNext = NULL;
		m_pSensor = pSensor;
	}
};
```

Реалізація однозв'язкового списку. У списку буде:
– Покажчик на перший вузол.
– Вказівник на останній вузол.
– Конструктор.
– Функція перевірки наявності вузлів у списку.
– Функція додавання елемента до кінця списку.
– Функція видалення вузла за заданим значенням.

```
class List
{
private:
	Item * m_pFirst;
	Item * m_pLast;
public:
	List()
	{
		m_pFirst = NULL;
		m_pLast = NULL;
	}

	bool IsEmpty()
	{
		return (m_pFirst == NULL);
	}

	Item * GetFirst()
	{
		return m_pFirst;
	}

	Item * GetLast()
	{
		return m_pLast;
	}

	void AddItem(Item * pItem)
	{
		if(IsEmpty())
```

```
		{
			m_pFirst = pItem;
			m_pLast = m_pFirst;
			m_pLast->m_pNext = NULL;
		}
		else
		{
			m_pLast->m_pNext = pItem;
			m_pLast = pItem;
			m_pLast->m_pNext=NULL;
		}
	}

	bool RemoveItem(Item * pItem)
	{
		Item * p;
		p=m_pFirst;
		while(p != NULL)
		{
			if(m_pFirst == pItem)
			{
				m_pFirst = m_pFirst->m_pNext;
				return true;
			}
			else if(p->m_pNext == pItem)
			{
				p->m_pNext = p->m_pNext->m_pNext;
				return true;
			}
			p=p->m_pNext;
		}
		return false;
	}
};
```

Реалізація класу ітератора. Основна ідея полягає в тому, щоб за доступ до елементів та спосіб обходу відповідав не сам список, а окремий об'єкт-ітератор. У класі ListIterator визначено інтерфейс доступу до елементів списку. Об'єкт цього класу відстежує поточний елемент m_pCurrent, тобто він має у своєму розпорядженні інформацію, які елементи вже відвідувалися.

```
class ListIterator
{
private:
	List * m_pList;
	Item * m_pCurrent;
public:
	ListIterator(List * pList)
	{
		m_pList = pList;
		m_pCurrent = NULL;
	}

	void First()
	{
		m_pCurrent = m_pList->GetFirst();
	}

	void Next()
	{
		m_pCurrent = m_pCurrent->m_pNext;
	}
```

```
        bool IsDone()
        {
                return (m_pCurrent == NULL);
        }

        Item * GetCurrentItem()
        {
                return m_pCurrent;
        }

};
```

У той час як вказівники використовуються для зберігання адреси пам'яті, якою б не була ця адреса, з контейнерами завжди використовується ітератор. Ітератор використовується для перебору елементів контейнера, при цьому елементи контейнера не потрібно зберігати в зарезервованій області пам'яті. Навіть якщо елементи розкидані по пам'яті, як, наприклад, у списку, ітератор все одно працюватиме.

Враховуючи, що вказівник завжди зберігає адресу пам'яті, її завжди можна перетворити на ціле число (яке є адресою). Більшість ітераторів не можуть бути перетворені на цілі числа. Далі наведено приклад використання створених класів.

```
int main(void)
{
        char str[16];

        Uart uart;
        Converter adc;

        LM35_Sensor T0(&adc, CHANNEL_0);
        LM35_Sensor T1(&adc, CHANNEL_1);
        LM35_Sensor T2(&adc, CHANNEL_2);
        LM35_Sensor T3(&adc, CHANNEL_3);

        Item listItem0(&T0);
        Item listItem1(&T1);
        Item listItem2(&T2);
        Item listItem3(&T3);

        List sensors;
        sensors.AddItem(&listItem0);
        sensors.AddItem(&listItem1);
        sensors.AddItem(&listItem2);
        sensors.AddItem(&listItem3);

        ListIterator iterator(&sensors);

        while(1)
        {
                iterator.First();
                while(iterator.IsDone() == false)
                {
                sprintf(str, "%d\r\n", iterator.GetCurrentItem()->m_pSensor->GetTemperature());
                iterator.Next();
                uart.SendString(str);
                }
                _delay_ms(1000);
        }
        }
```

Після компіляції проєкту і запуску системи на моделювання у вікні терміналу можна побачити виміряні значення температури, отримані з чотирьох датчиків:

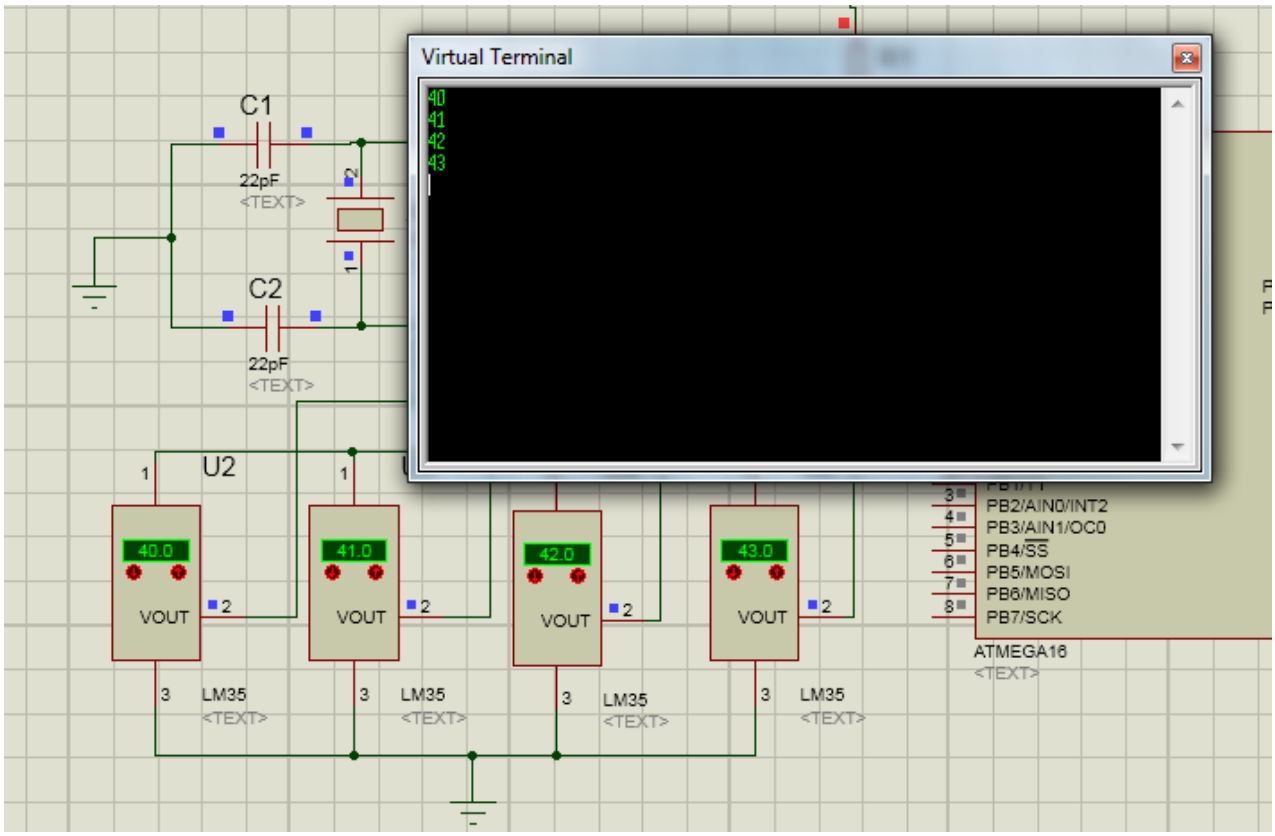


Рис. 16.

Ітератор надає клієнту кілька простих методів перебору елементів колекції. Це не лише спрощує доступ до колекції, а й захищає її дані від необережних чи зловмисних дій. Перш ніж створювати екземпляр класу ListIterator, необхідно мати список, що підлягає обходу. З об'єктом ListIterator можна послідовно відвідати всі елементи списку. Операція GetCurrentItem повертає поточний елемент списку, операція First ініціалізує поточний елемент першим елементом списку, Next робить поточним наступний елемент, а IsDone перевіряє, чи ми опинилися за останнім елементом, якщо так, то обхід завершено. Слід зазначити, що між ітератором та списком є тісний зв'язок, клієнт повинен мати інформацію, що він обходить саме список, а не якусь іншу агреговану структуру. Тому клієнт прив'язаний до конкретного способу агрегування.

Ланцюжок обов'язків (Chain of Responsibility).

Ланцюжок обов'язків – це поведінковий патерн проектування, який дозволяє передавати запити послідовно по ланцюжку обробників. Кожен наступний оброблювач вирішує, чи може він опрацювати запит сам і чи варто передавати запит далі ланцюжком.

Цей патерн зручно використовувати, наприклад, для обробки команд керування деякими виконавчими пристроями. При цьому самі команди можуть

надходити по універсальному асинхронному приймачу/передавачу у вигляді текстових рядків.

Як і багато інших поведінкових патернів, ланцюжок обов'язків базується на тому, щоб перетворити окремі поведінки в об'єкти. У даному випадку кожна перевірка перейде до окремого класу з єдиним методом виконання. Дані запиту, над яким відбувається перевірка, передаватимуться у метод як аргументи.

Найважливіше те, що цей паттерн пропонує зв'язати об'єкти обробників в один ланцюг. Кожен із них матиме посилання на наступний обробник у ланцюзі. Таким чином, при отриманні запиту обробник зможе не тільки сам щось зробити з ним, але й передати обробку наступному об'єкту в ланцюжку.

Передаючи запити в перший обробник ланцюжка, можна бути впевненим, що всі об'єкти в ланцюзі зможуть його обробити. При цьому довжина ланцюжка не має жодного значення. При цьому оброблювач не обов'язково повинен передавати запит далі, причому ця особливість може бути використана по-різному. Якщо виявиться, що дані, які передаються по ланцюжку, пошкоджені (наприклад, невірна контрольна сума в пакеті даних, бо невірний формат команди), тоді немає сенсу марнувати ресурси, якщо й так зрозуміло, що із запитом щось не так. На рис. 17 представлено структуру патерну.

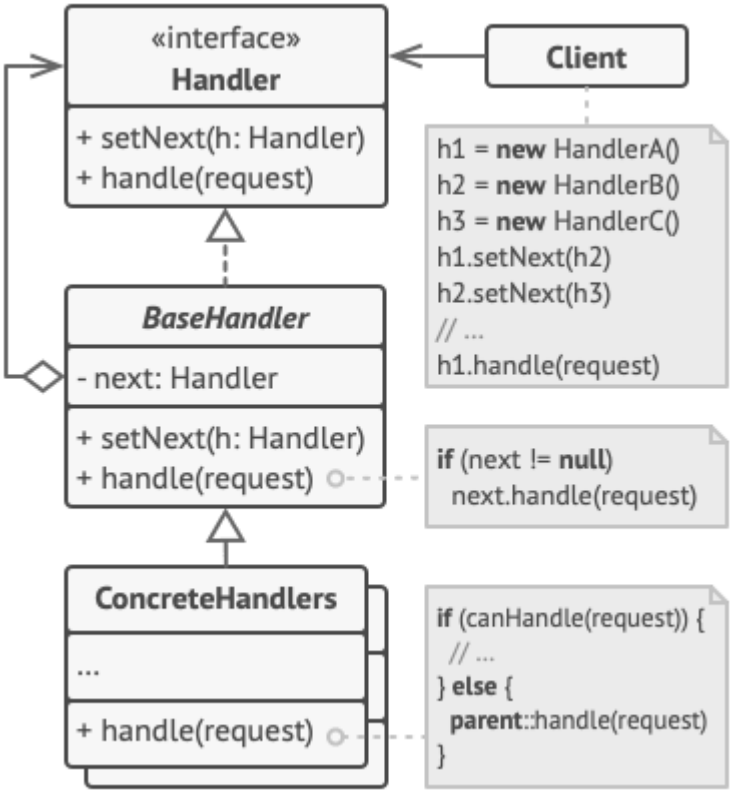


Рис. 17.

В даному прикладі цей патерн використовується при розробці системи керування виконавчими механізмами: світлодіодом, двигуном постійного струму та освітлювальною лампою. Команди на вмикання або вимикання надходять по UART та відповідно мають вигляд: LED ON, LED OFF, MOTOR ON, MOTOR OFF, LAMP ON, LAMP OFF. На рис. 18 представлено принципову схему системи.

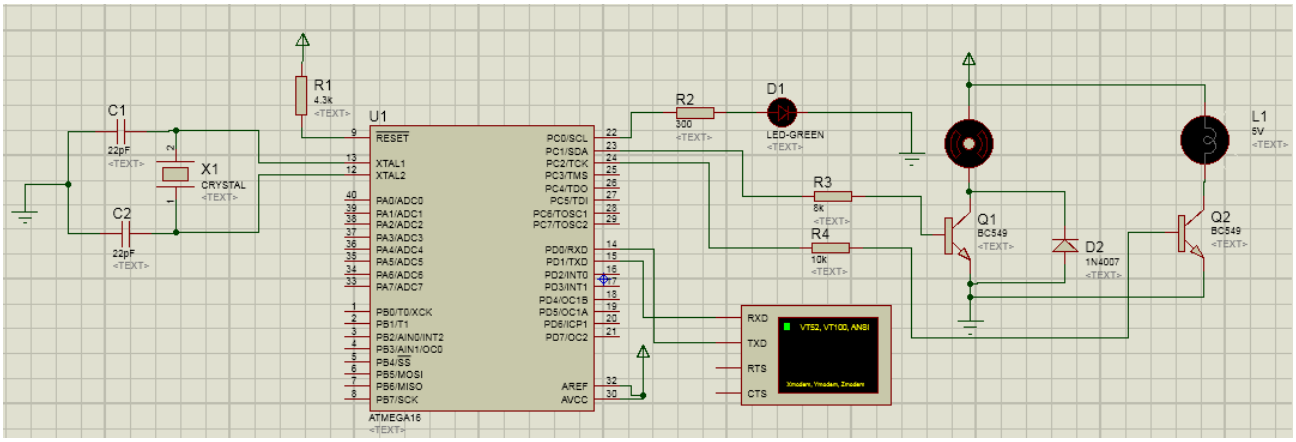


Рис. 18.

Текст програми представлено далі. Команда, що надходить по UART, зберігається в масиві символів. Ознакою завершення команди може бути символ CR (‘\r’) або LF (‘\n’).

```
#define F_CPU 8000000

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

#define MAX_STR_LEN 16
```

Оскільки керування виконавчими механізмами виконується через порт вводу/виводу мікроконтролера, тому необхідно створити клас, що надає високорівневий інтерфейс (набір функцій-членів класу) для керування станами виводів мікроконтролера. Цей клас може працювати з одним з чотирьох портів мікроконтролера ATMega16 – портами A, B, C та D. Для зручності в схемі використовуються лише порт C, виводи 0-2.

```
class Port
{
	private:
	char m_name;
	bool m_isOut;

	public:
	Port();
	Port(char name, bool isOut);

	void write(char value);
	char read();

	void set(char pos);
	void reset(char pos);
};
```

Конструктор за замовчуванням.

```
Port::Port()
{
	m_name = 'A';
	m_isOut = 0;
```

```
	DDRA = 0;
}
```

Перевантажений конструктор – параметрами виступають назва порта та режим його роботи (на вхід або на вихід).

```
Port::Port(char name, bool isOut)
{
	m_name = name;
	m_isOut = isOut;

	switch(m_name)
	{
		case 'A':
			DDRA = (isOut == 0)? 0 : 0xFF;
		break;
		case 'B':
			DDRB = (isOut == 0)? 0: 0xFF;
		break;
		case 'C':
			DDRC = (isOut == 0)? 0: 0xFF;
		break;
		case 'D':
			DDRD = (isOut == 0)? 0: 0xFF;
		break;
	}
}

void Port::write(char value)
{
	switch(m_name)
	{
		case 'A':
			PORTA = value;
		break;
		case 'B':
			PORTB = value;
		break;
		case 'C':
			PORTC = value;
		break;
		case 'D':
			PORTD = value;
		break;
	}
}

char Port::read()
{
	char value = 0;

	if(m_name == 'A')
	{
		value = PINA;
	}
	else if(m_name == 'B')
	{
		value = PINB;
	}
	else if(m_name == 'C')
	{
		value = PINC;
	}
	else if(m_name == 'D')
```

```
	{
		value = PIND;
	}

	return value;
}

void Port::set(char pos)
{
	switch(m_name)
	{
		case 'A':
			PORTA |= (1<<pos);
		break;
		case 'B':
			PORTB |= (1<<pos);
		break;
		case 'C':
			PORTC |= (1<<pos);
		break;
		case 'D':
			PORTD |= (1<<pos);
		break;
	}
}

void Port::reset(char pos)
{
	switch(m_name)
	{
		case 'A':
			PORTA &= ~(1<<pos);
		break;
		case 'B':
			PORTB &= ~(1<<pos);
		break;
		case 'C':
			PORTC &= ~(1<<pos);
		break;
		case 'D':
			PORTD &= ~(1<<pos);
		break;
	}
}
```

Клас для роботи з універсальним асинхронним приймачем/передавачем.

```
class Uart
{
	public:
	Uart()
	{
		UCSRA = 0;
		UCSRB = 0b00011000;
		UCSRC = 0b00000110;
		UBRRH = 0;
		UBRRL = 51;
	}

	void SendString(char * text)
	{
		for(unsigned int i = 0; i < strlen(text); i++)
		{
			UDR = text[i];
			while((UCSRA&(1<<UDRE))==0);
```

```
		}
	}
```

Цей метод зчитує дані, що надходять по UART, та розміщує їх у буфер. Якщо надійшов один з символів ‘\r’ або ‘\n’, метод припиняє свою роботу та прийнятий рядок зберігається у масиві символів (параметр методу).

```
	void GetString(char * str)
	{
		int index = 0;
		memset(str, 0, MAX_STR_LEN * sizeof(char));

		while(index < (MAX_STR_LEN-2))
		{
			while((UCSRA & (1 << RXC)) == 0);
			str[index] = UDR;
			if((str[index] == '\r') || (str[index] == '\n'))
			{
				break;
			}
			index++;
		}
		str[index] = '\0';
	}
};
```

Базовий обробник – опціональний клас, який дозволяє позбутися дублювання одного і того ж коду у всіх конкретних обробниках. Зазвичай цей клас має поле для зберігання посилання на наступний обробник у ланцюжку. Клієнт пов'язує оброблювачі в ланцюг, подаючи посилання на наступний оброблювач через конструктор або сетер (мутатор) поля. Також тут можна реалізувати базовий метод обробки, який просто перенаправляв би запит наступному обробнику, перевіривши його наявність.

Базовий обробник команди містить поля – вказівник на об’єкт порта, номер виводу, до якого підключени той чи інший виконавчий механізм, та вказівник на екземпляр наступного обробника команди для того, щоб мати можливість об’єднати всі обробники в ланцюг. Обробник визначає загальний всім конкретних обробників інтерфейс.

```
class Handler
{
protected:
	Port * m_pPort;
	char m_pinNum;
	Handler * m_pNext;
public:
	Handler(Port * pPort, char pinNum)
	{
		m_pPort = pPort;
		m_pinNum = pinNum;
		m_pNext = NULL;
	}

	void SetNext(Handler * pNext)
	{
		m_pNext = pNext;
	}
```

Зазвичай досить описати єдиний метод обробки запитів, але іноді (як в цьому прикладі) тут може бути оголошено метод виставлення наступного оброблювача. Цей метод і є обробником команди, він буде перевизначений в класах-нащадках.

```
	virtual void Handle(char * cmd)
	{
		if(m_pNext != NULL)
		{
			m_pNext->Handle(cmd);
		}
	}
};
```

Конкретні обробники містять код обробки запитів. При отриманні запиту кожен оброблювач вирішує, чи може він обробити запит, а також чи варто передати його наступному об'єкту. Найчастіше обробники можуть працювати власними силами і бути незмінними, отримавши всі необхідні деталі через параметри конструктора.

```
class LEDHandler : public Handler
{
public:
	LEDHandler(Port * pPort, char pinNum) : Handler(pPort, pinNum)
	{
		// Empty constructor
	}

	void Handle(char * cmd)
	{
		if(strcmp(cmd, "LED ON") == 0)
		{
			m_pPort->set(m_pinNum);
		}
		else if(strcmp(cmd, "LED OFF") == 0)
		{
			m_pPort->reset(m_pinNum);
		}
		else
		{
			Handler::Handle(cmd);
		}
	}
};


class MotorHandler : public Handler
{
	public:
	MotorHandler(Port * pPort, char pinNum) : Handler(pPort, pinNum)
	{
		// Empty constructor
	}

	void Handle(char * cmd)
	{
		if(strcmp(cmd, "MOTOR ON") == 0)
		{
			m_pPort->set(m_pinNum);
		}
```

```
            else if(strcmp(cmd, "MOTOR OFF") == 0)
            {
                  m_pPort->reset(m_pinNum);
            }
            else
            {
                  Handler::Handle(cmd);
            }
      }
};


class LampHandler : public Handler
{
      public:
      LampHandler(Port * pPort, char pinNum) : Handler(pPort, pinNum)
      {
            // Empty constructor
      }

      void Handle(char * cmd)
      {
            if(strcmp(cmd, "LAMP ON") == 0)
            {
                  m_pPort->set(m_pinNum);
            }
            else if(strcmp(cmd, "LAMP OFF") == 0)
            {
                  m_pPort->reset(m_pinNum);
            }
            else
            {
                  Handler::Handle(cmd);
            }
      }
};
```

Головна функція програми, в якій створюються екземпляри розроблених класів. Отримана команда зберігається у масиві символів і далі передається по ланцюжку. Клієнт може або сформувати ланцюжок обробників один раз, або перебудовувати його динамічно, залежно від логіки програми. Клієнт може надсилати запити будь-якому об'єкту ланцюжка, не обов'язково першому з них.

```
int main(void)
{
      char cmd[MAX_STR_LEN];
      Port port('C', true);

      Uart uart;
      LEDHandler ledHandler(&port, 0);
      MotorHandler motorHandler(&port, 1);
      LampHandler lampHandler(&port, 2);

      ledHandler.SetNext(&motorHandler);
      motorHandler.SetNext(&lampHandler);

      while(1)
      {
            uart.GetString(cmd);
            ledHandler.Handle(cmd);
      }
}
```

Після запуску процесу моделювання у вікні терміналу можна ввести одну з команд, як показано на рис. 19, та побачити зміну стану виконавчого механізму в залежності від надісланої команди.

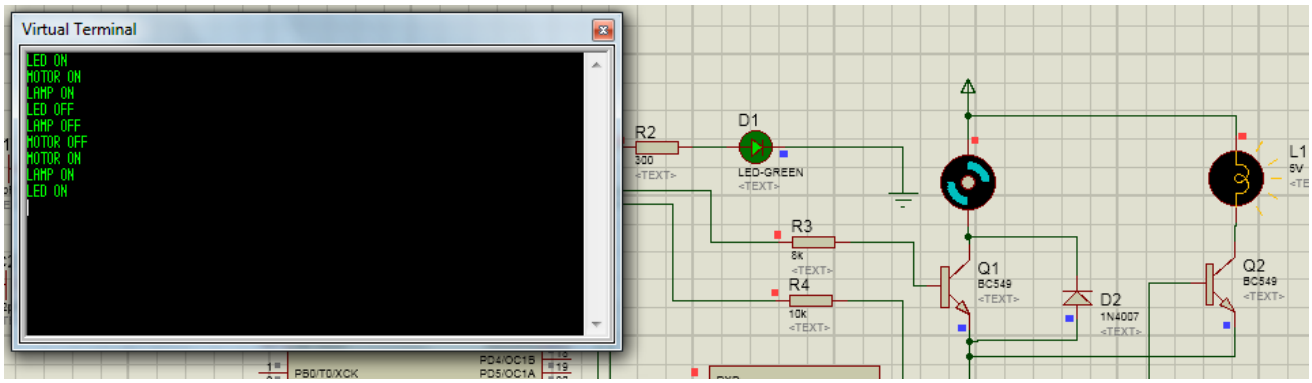


Рис. 19.

Отже, цей патерн може бути використаний коли програма повинна обробляти різноманітні запити декількома способами, але заздалегідь невідомо, які запити будуть надходити і які обробники їм знадобляться. За допомогою ланцюга обов'язків можна зв'язати потенційних обробників в один ланцюг і при отриманні запиту по черзі запитувати кожного з них, чи не хоче він обробити запит.

Завдання для самостійного виконання:

1. Розробити клас-адаптер, що виконує перетворення значення температури, отриманого з аналогового датчика LM35 (як в розглянутому прикладі), в градуси Кельвіна, та відправляє отримане значення по UART.

2. Реалізувати патерн Стратегія, додавши в наведений приклад ще один алгоритм, наприклад виведення на світлодіоди випадкових значень протягом 5 секунд зі зміною значення кожні 200 мс.

3. Оформити звіт, в якому навести розроблену програму. Завантажити звіт на гугл-диск.

ЛІТЕРАТУРА

1. Рябенький В.М. Програмовані електронні системи керування, збору та обробки інформації / В.М. Рябенький, О.О. Ушкаренко. – Миколаїв : НУК, 2021. – 490 с.

2. Рябенький В.М. Схемотехніка електронних пристроїв та систем. Том 3. Мікропроцесорна техніка / В.М. Рябенький, О.О. Ушкаренко, В.С. Буряк. – Миколаїв: Вид-во «Іліон», 2012. – 445 с.

3. Рябенький В. М. Схемотехніка: пристрої цифрової електроніки: підручник у 2 т. / В. М. Рябенький, В. Я. Жуйков, Ю. С. Ямненко, О. В. Борисов. – Київ : НТУУ «КПІ» Вид-во «Політехніка», 2015. – Т. 2. – 360 с.

4. Рябенький В.М., Фісун М.Т., Ушкаренко О.О. Схемотехніка електронних пристроїв та систем. Том 2. Прикладна теорія цифрових автоматів. Миколаїв: Вид-во ЧДУ ім. Петра Могили, 2010. – 380 с.

5. Якименко Ю. І. Мікропроцесорна техніка / Ю. І. Якименко, Т.О. Терещенко, Є. І. Сокол, В. Я. Жуйков, Ю. С. Петергеря. – Київ : Політехніка, 2004. – 440 с.

6. Рябенький В.М., Жуйков В.Я., Гулий В.Д. Цифрова схемотехніка: Навч. Посібник. – Львів: «Новий Світ-2000», 2009. – 736 с.

7. Фрімен Ерік. Патерни проєктування / Ерік Фрімен, Елізабет Робсон, Берт Бейтс, Кеті Сієрра. – Київ : Фабула, 2020. – 672 с.

8. Майа Пошевай. Програмування вбудованих систем на C++ 17 / Пошевай Майа. Київ : ДМК-Пресс, 2020. – 394 с.

9.Рябенький В.М. Схемотехніка електронних пристроїв та систем. Том 6. Апаратно-програмні засоби відображення інформації / В.М. Рябенький, О.О. Ушкаренко. – Миколаїв: Вид-во «Іліон», 2013. – 464 с.

10. Документація розробників програмних ресурсів мікропроцесорних систем. Режим доступу: https://embeddedexpert.io/

11. Публікації розробників вбудованих електронних систем. Режим доступу: https://www.sciencedirect.com/topics/computer-science/embedded-systems

12. Документація по мовам програмування C/C++. Режим доступу: https://www.eolymp.com/uk/blogs/posts/26

13. Уроки програмування на C++. Режим доступу: https://ravesli.com/uroki-cpp/

14. Основи програмування на C++. Режим доступу: https://purecodecpp.com/

15. Огляд і основи мови програмування C++. Режим доступу: http://www.znannya.org/?view=Cpp_basics

16. Вступ до програмування мовою C++. Режим доступу: http://csc.knu.ua/en/library/books/belov-24.pdf