НАВЧАЛЬНО-МЕТОДИЧНА ЛІТЕРАТУРА

Тернопільський національний технічний університет імені Івана Пулюя

Кафедра автоматизації технологічних процесів і виробництв

Лабораторна робота № 17 з курсу ”Проектування систем автоматизації”

Об'єктно-орієнтоване програмування контролерів сімейства Arduino

Тернопіль 2016

Методичні вказівки до лабораторної роботи № 17 “Об'єктно-орієнтоване програмування контролерів сімейства Arduino” з курсу "Проектування систем автоматизації". Шкодзінський О.К., Пісьціо В.П., Медвідь В.Р., Тернопіль: ТНТУ, 2016 - 19 с.

Для студентів напряму: 6.050202 "Автоматизоване управління технологічними процесами"

Автори: Шкодзінський О.К., Пісьціо В.П., Медвідь В.Р.

Методичні вказівки розглянуті, схвалені і затверджені на засіданні кафедри автоматизації технологічних процесів і виробництв (протокол № 4 від 21 листопада 2016 року).

Тема роботи

Об'єктно-орієнтоване програмування контролерів сімейства Arduino

Мета роботи

Ознайомитись із об'єктно-орієнтованим програмуванням контролерів сімейства Arduino а також із використанням переривань

Об’єктна модель у Arduino

У попередній лабораторній роботі було побудовано програму для виконання простих функцій вводу-виводу та обробки даних. Проте подальша модифікація програми викликає деякі труднощі

- ◊ Програмний блок для обробки сигналів розміщується поруч із іншими блоками та погіршує читання програми.
- ◊ Необхідно пам’ятати для чого кожна із глобальних змінних.
- ◊ Якщо необхідно використати декілька кнопок, що виконують схожі дії необхідно для кожної із них створити свої процедури вводу даних, обробки їх, тощо.
- ◊ При кожному використанні програмного блоку прийдеться повторно переписувати програмний код.

Для уникнення таких проблем у мові програмування Arduino використовують об’єктно орієнований підхід. При цьому можна використовувати, власні типи, структури даних, масиви і класи. Також є можливість використання посилань на дані і вказівників.

Декларування типів за допомогою typedef

Для опису (декларування) власних типів даних можна використовується ключове слово typedef. Фактично typedef е свого роду макросом, що присвоює назві типу його опис.

Опис власного типу у такому випадку буде наступний

```
typedef <опис типу> <назва типу>
```

Наприклад

```
typedef unsigned char Tdate;  //визначити тип Tdate як
//  безнаковий символьний тип
typedef byte Tval[3];  //визначити тип Tval як масив із
//  3 елементів типу byte
```

Описані за допомогою typedef типи можна використовувати аналогічно до вбудованих типів, наприклад, при описі змінних.

Вказівники

Вказівники призначені для збереження адреси комірки у пам'яті, наприклад для передавання у функцію адреси масиву, структури даних, що будуть змінюватись "на місці", а також для визначення списків і подібних структур даних. Насправді у Arduino досить рідко виникає потреба у таких структурах даних, проте можливість їх використання наявна.

Визначення змінної-вказівника здійснюється записом

```
base_type *pvar;
```

змінна pvar отримує тип вказівника на base_type.

Проте, такий опис заплутує опис змінних, тому можна скористатись typedef і записати опис типів таким чином:

```
typedef  base_type * pbase_type;  //визначаємо новий тип
// вказівника на base_type
pbase_type pvar; //визначаємо змінну типу
//pbase_type = вказівника на base_type
```

При другому записі одночасно визначається спеціальний тип pbase_type, що може бути використаний для зручного представлення змінних.

Оператор & повертає адресу змінної, а оператор * використовується для доступу до значення змінної, на котру посилається вказівник:

```
typedef int* Pint;
```

```
int sourceNum1 = 100;
int sourceNum2 = 200;
Pint pNum1 = &sourceNum1; // визначимо вказівник pNum1 і
//задамо йому значення адреси змінної sourceNum1
int* pNum2 = &sourceNum2; // визначимо вказівник pNum2 і
// задамо йому значення адреси змінної sourceNum2
*pNum1 = 2; //У комірку пам'яті на котру посилається змінна
//  pNum1 запишемо 2. Так як
pNum1 містить адресу sourceNum1 значення 2 буде присвоєно sourceNum1
pNum1 = pNum2; //Змінній pNum1 присвоємо адресу із pNum2
*pNum1 = 3; //Тепер 3 запишеться за новою адресою, що знаходиться у
// pNum1 тобто sourceNum2 стане рівним 3
sourceNum1 =  (*pNum2)  + sourceNum2; // А тепер додамо вміст комірки
на котру посилається pNum2 та sourceNum2 результат додавання буде рівний 6.
```

Для вказання що змінна не посилається на жодну адресу традиційно використовують нульовий вказівник NULL

```
pNum1  = NULL
```

Також визначений вказівник на комірку пам'яті що не має типу. Він визначається так

```
void * p
```

Тип void не містить жодної комірки пам'яті. Значення з будь-якого вказівника може бути присвоєно змінній типу void *, а зворотне присвоювання можливе лише при явному приведенні типів.

Для вказівників визначені операції порівняння і адресна арифметика. Операції порівняння виду

```
pNum1 == pNum2  чи  pNum1 != pNum2
```

часто використовують для послідовного перебору елементів масиву в циклі, операції виду pNum1 == NULL використовують при контролі вмісту вказівників.

Для вказівників визначені команди інкремент та декремент. На відміну від простих типів де команда інкремент та декремент змінювала значення змінної на 1 команди інкремент та декремент вказівників змінюють значення на довжину змінної на котру посилається цей тип вказівника. Наприклад припустимо є фрагмент коду

```
pNum1 = &sourceNum1; // вказівник pNum1 має значення адреси змінної
sourceNum1
pNum1++  // Після виконання команди pNum1 буде зберігати наступну
адресу пам'яті відразу за змінною sourceNum1 у даному випадку там знаходиться
sourceNum2
pNum1--  // Після виконання команди pNum1 буде зберігати адресу адресу
пам'яті ле знаходиться змінна sourceNum1
```

Вказівниками досить складно керувати. Достатньо легко записати в вказівник невірне значення. Тому при роботі із вказівниками варто провести ініціалізацію вказівника відразу ж при його об'явленні, а також не змішувати вказівники із звичайними змінними (наприклад не варто записувати у коді int x, *p, y, *y_ptr, а ще краще визначати тип вказівника через typedef.

Переліки

Ключове слово enum використовується для оголошення переліку - окремого типу, який складається з набору іменованих констант, що називається списком переліку. Зазвичай перелік визначають відразу у основній частині програми, хоча можна визначити їх і у середині опису функцій, чи класів. Формат опису переліку наступний:

```
enum {AAA,BBB,CCC};
```

За замовчуванням нумерація елементів відбувається з нуля, тобто AAA еквівалентно константі 0, BBB = 1 і т.д. У випадку необхідності нумерація може бути почата із будь-якого значення. Як бачимо, нумерація у переліку може бути перервана і змінена, єдине правило -

всі елементи мають бути у порядку збільшення. Якщо елемент не має явно заданого номера він отримує наступний.

У наступному прикладі Sun отримає номер1, Mon - 2, Tue -4, а Sat - 33.

```
enum Days: byte { Sun = 1, Mon, Tue = 4, Wed = 8, Thu = 16, Fri = 32,
Sat };
```

Об'єднання union

Об'єднання (union) це тип, в якому всі члени використовують едину область пам'яті. Це означає, що у будь-який момент часу об'єднання не може зберігати більше одного об'єкту зі списку своїх членів. Незалежно від кількості членів об'єднання, воно використовує, об'єм пам'яті необхідний для збереження свого найдовшого елемента.

Вони можуть бути корисні для економії пам'яті та при приведенні типів об'єктів, наприклад при побайтовому читанні структури даних і подальшої обробки даних згідно вказаного формату.

Опис об'єднання визначається наступним синтаксисом

```
union [ім'я]  { іменований список членів };
```

Декларація починається словом union і містить список членів , наприклад:

```
union RecordType    // Declare a simple union type
{   char   ch;
    int    i;
    long   l;
    float  f;
    double d;
    int *int_ptr; };
```

Використання об'єкта

```
    RecordType t;  // оголошення змінної типу RecordType
    t.i = 5; // запис змінній t значення 5
    t.f = 7.25 // запис у змінну t позначення типу float
```

Зверніть увагу, що після виконання цих дій змінна t.i набуде значення, котре рівне бітовому представленню частини числа 7.25.

Програма "не знає" який тип даних був поміщений у об'єднання і якщо програміст записував дані одного типу, а зчитував іншого програма не покаже жодних помилок. Якщо наперед не відомо дані котрого типу знаходяться у об'єднанні слід використати окрему змінну, котра буде вказувати тип даних, що зберігаються. Є можливість зробити об'єднання безіменним:

```
    union { unsigned byte  iA;
        char aA; };
```

При цьому доступ спрощується, але програма стає більш заплутаною. Доступ до змінної:

```
  aA =  'A';  // змінна  aA стає рівною 'A' і одночасно змінна iA
  // набуває значення, котре відповідає коду символу 'A'
```

Структури даних

Структури (struct) даних призначені для отримання складних типів із декількома компонентами. На відміну від масивів структури можуть містити компоненти різних типів, а на відміну від об'єднань - всі компоненти зберігаються у власних комірках пам'яті. Компоненти структури називають полями.

Структури являють собою найбільш просту реалізацію об'єктного підходу при програмуванні, на відміну від класів, усі поля структуру доступні для зміни, а сама структура не може містити функцій, конструкторів та деструкторів.

Формат опису структури показаний нижче

```
struct str_name  //опис структури названої str_name
{
     int   member_1;    //перше поле
```

```
    float member_2;    //друге поле
    char  member_3[5]; //третє поле (масив символів)
};
// Опис змінної без ініціалізації
struct str_name struct0;
// Опис змінної з ініціалізацією (загальна форма)
struct str_name struct1 = {1, 3.1416f, "doit" };
// Опис змінної з ініціалізацією кожного поля
struct str_name struct3 =
{.member_1=2, .member_2=3.1416f, .member_3="doit"};
```

Опис структур допускає рекурсію - тобто в описі структури може знаходитись посилання на об'єкт типу структури. Крім того структура може містити у вигляді полів інші структури. Доступ до полів структури виконується за допомогою вказування імені змінної і імені поля, що розділені крапкою. Наприклад

```
struct3.member_1 = 33; // присвоїти полю member_1 структури
                       //struct3  значення 33.
```

Класи в Ардуїно

На відміну від структур класи дозволяють створювати більш зручні типи об'єктів. В класи входять властивості і методи. Властивості це данні, що характеризують об'єкт класу, а методи – функції, що виконують дії над об'єктом. Фактично:

- ◊ властивості класу - змінні об'єкта.
- ◊ методи класу - функції об'єкта.

В основу поняття "клас" покладено той факт, що "над об'єктами можна здійснювати різні операції". Властивості об'єктів описуються за допомогою полів класів, а дії над об'єктами описуються за допомогою функцій, які називаються методами класу. Клас має ім'я, складається з полів, званих членами класу і функцій - методів класу. Члени класу доступні із будь-якого метода класу і їх не потрібно передавати у якості параметрів функцій-методів.

Опис класу має наступний формат

```
class name //  name – ім'я класу
{
private: // Опис закритих членів і методів класу
protected: // Опис захищених членів і методів класу
public: // Опис відкритих членів і методів класу
}
```

Доступ до полів і методів здійснюється через оператор взяття поля ".". На відміну від полів структури, котрі доступні завжди і всім у класах можуть бути поля і методи різного рівня доступу:

- ◊ **public** Необмежений доступ
- ◊ **protected** Доступ обмежений класом чи похідними типами
- ◊ **private** Доступ обмежений лише типом, що описується.

Якщо у описі елементів класу відсутні вказування метода доступу то члени і методи класу вважаються закритими (private). Зазвичай методи описують за межами класу, проте можливий опис методів і у середині опису класу.

Після опису класу необхідно описати змінну вказаного типу. Наприклад

```
name_class name;
```

Де name_class - ім'я класу, name – ім'я змінної.

У подальшому змінну типу class будемо називати "об'єкт" чи "екземпляр класу". При об'явленні змінної вказаного типу може здійснюватись виклик спеціальної функції - конструктора, що ініціалізує поля класу.

Для більш повного опису класів розглянемо приклад: клас complex для роботи із комплексними числами

У класі complex будуть члени:

float Re – дійсна частина комплексного числа;
float Im – уявна частина комплексного числа.

методи класу:

float modul() – функція обчислення модуля комплексного числа;
double arg() – функція обчислення аргументу комплексного числа.

Функцію modul() опишемо у середині класу, а функцію arg() лише об'явимо у середині класу, а опишемо пізніше.

Зауважмо, що якщо функція, що є методом класу, (у тому числі конструктором чи деструктором) описується за межами опису класу перед її іменем записується назва класу і символи "::".

```
class complex
{ public:
  float Re, Im;;
//Метод класса
  float Amp()
{ return sqrt(Re*Re+Im*Im); } // опис метода при описі класу
  float Arg();//об'явлення метода при описі класу
}
float complex::Arg();  // опис метода класу за межами опису класу
{ return atan2(Im,Re);};
```

Використання класу. Визначимо змінні

```
complex C1;
float A,Ph;
Присвоюємо var1 комплексне значення
C1.Re = 3.5;//дійсна частина
C1.Im = -.5;//уявна частина
A = C1.Amp();//модуль комплексного числа
Ph = C1.Arg();//аргумент комплексного //числа
```

Використання відкритих членів і методів дозволяє отримати повний доступ до елементів класу: що не дозволить змінити реалізацію класу у подальшому і дозволяє внести помилку у функціонування методів класу. Наприклад, якщо у подальшому виявиться, що значно зручніше користуватись внутрішнім представленням комплексних чисел через аргумент та модуль всі підпрограми роботи із комплексними числами прийдеться переписувати.

Тому загальним принципом побудови класів є відділення реалізації класу від його представлення: «Чим менше відкритих даних про клас тим краще».

Конструктори

Конструктори призначені для створення об'єктів заданого типу. Ім'я конструктора співпадає із іменем класу, конструктор автоматично запускається при створенні об'єкта, (наприклад при оголошенні змінної).

Конструктор фактично є звичайною функцією, він не повертає значення, але при описі конструктора не слід вказувати тип значення `void`.

Для прикладу добавимо у створений вище клас конструктори із різним числом аргументів. Для цього у описі запишемо:

```
class complex
{public: //Прототипи конструктора класу.
...
complex(); //конструктор за замовчуванням
complex(float x); //конструктор з одним аргументом
complex(float x, y); //конструтор з двома аргументами
...};
// конструктори класу complex
```

```
complex::complex() {Re = 0; Im = 0;}
complex::complex(float x, y) {Re = x; Im = y;}
complex::complex(float x)    {Re = x; Im = 0;}
```

Використання трьох конструкторів дозволяє створювати комплексні числа різними методами. Зауважимо, що якщо використати конструктор із аргументами за замовчуванням число конструкторів може бути зменшена до одного. Для цього в описі конструктора необхідно записати

```
complex(float x=0, float y=0); //конструктор з двома
                               //аргументами.
```

У випадку виклику конструктора із одним аргументом чи без них замість відсутніх значень будуть підставлені значення задані при описі функції. У даному випадку - нулі.

Ще одним видом конструктора є конструктор копіювання, що дозволяє створювати копію об'єкта. Це актуально тоді, коли необхідні два об'єкти з одними і теми ж значеннями членів класів. Синтаксис заголовка конструктора копіювання наступний:

```
class_name(class_name2 & name);
```

class_name(— ім'я конструктора і ім'я класу, name — об'єкт із котрого знімається копія типу class_name2.

Конструктори копіювання використовують тоді, коли при копіюванні об'єктів слід виконати нетривіальні дії: наприклад необхідно підрахувати скільки об'єктів створюється, виділити пам'ять на копію, тощо. У випадку відсутності конструктора копіювання при копіюванні даних здійснюється побайтне копіюванні полів об'єкта.

Деструктори

Деструктори призначені для знищення об'єктів. Вони викликаються при виході із функції та при вивільненні пам'яті від об'єкта. Зазвичай вони виконують нетривіальні дії для знищення об'єкта: вивільнення обраної пам'яті, відключення апаратних ресурсів, тощо. Якщо клас не має явно описаного деструктора виконується деструктор за замовчуванням, котрий не виконує жодних дій.

Деструктор описується за допомогою синтаксису

```
~class_name();
```

деструктор не має аргументів і не може мати ім'я відмінне від стандартного.

Статичні члени

Ключове слово static у класах використовується для створення локальної змінної єдиної для всього класу. Звичайні змінні створюються для кожного об'єкту відповідного класу, а статична змінна єдиною для всіх об'єктів заданого класу і створюється на початку роботи програми навіть перед створенням першого об'єкта. Використовуючи статичні члени можна з легкістю організувати лічильник кількості об'єктів.

Масив об'єктів

Так як і інші типи, об'єкти класів можуть об'єднуватись у масиви. При створенні масиву для кожного елемента масиву буде автоматично викликатись конструктор, проте його виклик буде проходити без параметрів. Немає зручного метода вказати різні параметри для конструкторів масиву об'єктів.

Приклад класу в Arduino

Створимо клас для аналізу кнопки. Назвемо його Button. Нам необхідні такі властивості ознака натиснення кнопки, ознака кліка по кнопці, лічильник підтвердження сталого стану. До вказаних вище змінних добавимо номер виводу, до котрого під'єднана кнопка, а також змінну timeButton, що вказує час мінімального стабільного часу при спрацюванні кнопки.

Оформимо згадані змінні як властивості класу.

```
class Button {
  public:
boolean flagPress;    // ознака натиснення кнопки
```

```
boolean flagClick;    // ознака кліка по кнопці
void scanState();     // метод перевірки стану сигналу
void setPinTime(byte pin, byte timeButton); // метод
//встановлення номера виводу

  private:
byte  buttonCount;    // лічильник підтвердження сталого
// стану
byte timeButton;      // час мінімального стабільного стану
// кнопки, на котру має бути реакція
byte _pin;            // номер виводу
 };
```

Змінні timeButton и pin приїдеться заявляти як аргументи методу для встановлення значень. Тому добавлено _ перед іменами, щоб відрізняти аргументи методу і змінні класу. Також необхідно добавити методи для перевірки стану сигналу кнопки - scanState. Метод void scanState() не має аргументів і нічого не повертає.

Зміні номер виводу і час підтвердження були зроблені приватними, тому необхідна функція встановлення їх значень - метод void setPinTime(byte pin, byte timeButton) що має два аргументи: номер виводу і час підтвердження стабільного стану кнопки.

Код scanState()

```
// метод перевірки стану кнопки
// flagPress= true  - натиснута
// flagPress= false - не натиснута
// flagClick= true - була натиснута (клік)
void Button::scanState() {
 if ( flagPress == (! digitalRead(_pin)) )
     {  // стан сигналу лишався попереднім
        _buttonCount= 0;  // скид лічильника станів сигналу
     }
else { // стан сигналу змінився
     _buttonCount++;   // +1 до лічильника стану сигналу
     if ( _buttonCount >= _timeButton )
      {// стан сигналу не змінився за заданий час
       // стан сигналу став стійким
      flagPress= ! flagPress; // інверсія ознаки стану
      _buttonCount= 0;  // скид лічильника стану сигналу
       if ( flagPress == true )
       {flagClick= true;} // ознака кліка
    }
  }
}
```

Метод setPinTime(byte pin, byte timeButton) - ще простіше - він копіює аргументи у приватні поля та встановлює режим виводів.

```
void Button::setPinTime(byte pin, byte timeButton)  {
  _pin= pin;
  _timeButton= timeButton;
  pinMode(_pin, INPUT_PULLUP);  // визначаємо вивід як вхід
}
```

Використовуємо клас.

```
Button button1;   // створюємо об'єкт button1 типу  Button
```

Для звертання до членів класу з будь-якого місця програми необхідно використовувати ім'я об'єкту, крапку і ім'я властивості

```
button1.flagClick= false;   //змінній  flagClick присвоїти  false
```

```
button1.scanState();       // виклик методу scanState(), об'єкта
button1
button1.setPinTime(12, 20); // виклик метода setPinTime (), об'єкту
button1 з параметрами 12 та 20
```

Перевірка стану кнопки button1 із будь-якого місця програми буде виглядати так:

```
if ( button1.flagPress == true )  {// кнопка натиснута }
```

Приклад основної програми із використанням об'єктного підходу

```
//*  Кожне натиснення кнопки змінює стан світлодіода/
#define LED_PIN 13     // світлодіод підключений до виводу 13
#define BUTTON_PIN 12  // кнопка підключена до виводу 12
// Опис класа обробки сигналів кнопок
class Button {
  public:
    boolean flagPress;//ознака натиснення кнопки
    boolean flagClick;//ознака що кнопка була натиснута раніше
    void scanState(); // метод перевірки стану сигналу
    void setPinTime(byte pin, byte timeButton);
//метод встановлення номера виводу і часу підтвердження
  private:
    byte  _buttonCount;
       // лічильник підтверджень стабільного стану
    byte  _timeButton;// час підтвердження стану кнопки
    byte  _pin;       // номер виводу
};
boolean ledState; // змінна стану світло діоду
Button button1;   // створення об'єкта типу Button button1
void setup() {
  pinMode(LED_PIN, OUTPUT); // визначаємо вивід  13 як вихід
button1.setPinTime(BUTTON_PIN, 15);
// виклик метода встановлення параметрів button1:
// номер виводу 12, число підтверджень 15
// нескінченний цикл із періодом 2 мс
void loop() {
  button1.scanState();  // виклик методу сканування кнопки
  // керування світлодіодом
  if ( button1.flagClick == true ) {
    // якщо було натиснення кнопки
    button1.flagClick= false;  //скидання стану кліка
    ledState= ! ledState;      //інверсія стану світлодіода
    digitalWrite(LED_PIN, ledState);
                               // вивід стану світлодіода
  }
  delay(2);  // затримка на 2 мс
}
// метод перевірки стану кнопки
// flagPress= true  - натиснута
// flagPress= false - відпущена
// flagClick= true - була натиснута
void Button::scanState() {
 if ( flagPress == (! digitalRead(_pin)) ) {
     // стан сигналу лишився попереднім
     _buttonCount= 0;  // скид лічильника стану сигналу
  }
```

```
  else {    // стан сигналу змінився
     _buttonCount++;   // +1 до лічильника стану сигналу
     if ( _buttonCount >= _timeButton ) {
      // стан сигналу не змінювався заданий час
      // стан сигналу став стійким
      flagPress= ! flagPress; // інверсія ознаки стану
      _buttonCount= 0;  // скид лічильника стану сигналу
      if ( flagPress == true ) flagClick= true;
       // ознака кліка
     }
  }
}
// метод встановлення номера виводу та часу підтвердження
void Button::setPinTime(byte pin, byte timeButton)  {
  _pin= pin;
  _timeButton= timeButton;
  pinMode(_pin, INPUT_PULLUP);  // визначаємо вивід як вхід
}
```

Бібліотеки для Ардуїно.

Вище був створений клас Button – кнопка, але він лишився не зручний при для використання у подальших проектах - кожен раз у нову програму необхідно копіювати опис класу та методів, що може привести до помилок. Значно краще оформити клас у вигляді спеціальної бібліотеки. Бібліотека повинна мати як мінімум два файли:

- ◊ файл заголовків (розширення .h)
- ◊ файл із вихідним кодом (розширення .cpp).

У першому файлі знаходиться опис класу , змінні та константи. Коду програми у там звичайно немає. Другий файл містить програмний код методів, котрі були описані у першому файлі.

Файл Button.h

Зазвичай у файлі заголовків крім опису коду вказують текстову інформацію про використання бібліотеки, тощо.

Решту вмісту h-файла необхідно заключити у конструкцію:

```
// перевірка, що бібліотека ще не підключена
#ifndef Button_h  // якщо бібліотека Button не підключена
   // приводимо опис бібліотеки і .....
#define Button_h  // визначаємо спеціальну константу, що вказує на
підключення бібліотеки
#endif
```

У середині конструкції слід внести директиву:

```
#include "Arduino.h"
```

Директива #include каже компілятору включити у код програми текст із файла, ім'я котрого слідує після директиви. У даному випадку буде включений файл Arduino.h, що містить стандартні константи та змінні для Ардуїно. У звичайних програмах він добавляється автоматично, а для бібліотек - вказується явно у разі необхідності.

Лишилось добавити опис класу Button. Повністю файл Button.h наведений у додатку Д1.

Файл Button.cpp.

На початку файла розмістимо додаткову інформацію про бібліотеку і далі запишемо директиви #include для включення функцій Ардуіно і файла Button.h.

```
#include "Arduino.h"
#include "Button.h"
```

А потім у бібліотеку запишемо методи вказаного вище класу.

Повністю файл Button.cpp наведений у додатку 2.

Розміщення файлів бібліотек і оформлення

Для розміщення бібліотек зазвичай використовують каталог (папку) libraries у корені дерева каталогів Арудуіно. Наприклад, якщо середовище програмування Ардуіно встановлене у каталог "C:\Program Files\Arduino" бібліотеки розміщаються у каталог "C:\Program Files\Arduino\libraries"

Для кожної бібліотеки слід створити свій каталог (папку) із іменем, шо співпадає із назвою файла із розширенням h, а каталог має містити файли бібліотеки. Тобто для використання бібліотеки необхідно створити каталог Button у каталозі "C:\Program Files\Arduino\libraries" і помістити туди всі вищезгадані файли бібліотеки. Для використання бібліотеки необхідно в початку програми вказати

```
#include <Button.h>
```

Переривання

У реальній програмі необхідно одночасно здійснювати багато різних дій. Крім того часто необхідно проводити реакцію на зовнішні події, вимірювати інтервали часу, тощо. Всі такі операції виконуються циклічно і паралельно, із різними періодами циклів, жодну із них не можна призупинити.

Зручним методом роботи із такими подіями буде режим переривань. У такому режимі за сигналом запиту переривання робота основної програми призупиняється, а управління передається підпрограмі обслуговування переривання. Після обслуговування переривання управління передається основній програмі і вона виконується із місця зупинки. Для основної програми виконання підпрограми обслуговування переривань невидиме. Короткий час роботи такої підпрограми не буде впливати на виконання основної програми.

Контролери ATMega мають багато різних джерел переривань частина котрих використана у середовищі Ардуіно.

Наприклад можна так настроїти систему, щоб кожні 2 мс викликалась підпрограма обслуговування переривання таймера 2, що запускає підпрограму користувача. Встановлення режиму і часу спрацювання таймера Ардуїно проводиться зазвичай через апаратні регістри мікроконтролера. Для полегшення програмування існує безліч бібліотек, що полегшують використання переривань.

Наприклад бібліотека MsTimer2 призначена для конфігурування апаратного переривання від таймера 2 мікроконтролера. Она має всього три функції:

```
MsTimer2::set(unsigned long ms, void (*f)())
```

Функція встановлює час переривання у мс. З таким періодом буде викликатись функція обробник переривання f(). Він повинен були об'явлений як void (не повертає нічого) та не мати аргументів.

```
MsTimer2::start()
```

Функція дозволяє переривання від таймера.

```
MsTimer2::stop()
```

Функція забороняє переривання від таймера.

Проста програма з паралельною обробкою сигналу кнопки

Тепер напишемо просту програму з однією кнопкою та світлодіодом, що виконує аналогічні фунції, але використовує переривання.

В функції setup() задаємо час циклу переривання за таймером 2 мс і вказуємо ім'я обробника переривання timerInterrupt. Функція обробки сигналу кнопки button1.scanState() викликається в обробнику переривання таймера кожні 2 мс.

```
// Натискання на кнопку змінює стан світлодіода
#include <MsTimer2.h>
#include <Button.h>
#define LED_1_PIN 13     // світлодіод підключений до к виводу 13
#define BUTTON_1_PIN 12  // кнопка підключена до виводу 12
Button button1(BUTTON_1_PIN, 15);  // створення об'єкту - кнопки
```

```
void setup() {
  pinMode(LED_1_PIN, OUTPUT);      // визначаємо вивід світлодіода як
вихід
  MsTimer2::set(2, timerInterupt);
// задаємо період переривання від таймера в 2 мс
  MsTimer2::start();              // дозволяємо переривання
}
void loop() {
//керування світлодіодом
  if ( button1.flagClick == true ) {
    // був клік кнопки
    button1.flagClick= false;         // скид ознаки
    digitalWrite(LED_1_PIN, ! digitalRead(LED_1_PIN));
// інверсія стану світлодіоду

  }
}
// обробка переривання
void  timerInterupt() {
  button1.scanState();  // виклик методу очікування стабільного стану
кнопки
}
```

Таким чином, стан кнопки оброблюється паралельним процесом. А в основному циклі програми перевіряється ознака кліка кнопки та змінюється стан світлодіода.

Завдання

Написати програму для автоматизації вимірювання характеристики деякого приладу.

Необхідно визначати із періодом 10 мс значення напруги на виводі вказаному викладачем. Дані, що вимірюються направляються за допомогою послідовного інтерфейсом у ПК. Процес вимірювання починається при натиснутій кнопці S1.

Кнопку та аналоговий інтерфейс описати із використанням власної бібліотеки. Моменти вимірювань та процес вимірювання аналогового сигналу проводити із використанням переривання.

Література

1. Соммер У. Программирование микроконтроллерных плат Arduino/Freeduino. — СПб.: БХВ-Петербург, 2012. — 256 с. ил — (Электроника)

2. Юревич Е.И. Проектирование технических систем. СПб.Питер 2001. 96 с.

Додаток 1 Файл Button.h

```
    /* Button.h - бібліотека для обробки сигналів кнопок  і сигналів інших
компонентів  У процесі роботи повинен регулярно викликатись метод:
      void  scanState(); // метод перевірки стабільного стану сигналу
    У результаті формуються ознаки:
 - при натиснутій  кнопці flagPress= true
 - при відпущеній  кнопкц flagPress= false
 - при натисканні flagClick= true
    Об'єкт типу Button при створенні має параметри:
 - номер виводу, до котрого підключена кнопка чи сигнал
 - час обробки (у періодах виклику scanState())
    */
    // перевірка, що бібліотека не підключена
    #ifndef Button_h // якщо  бібліотека Button не підключена і
    // константа Button_h не визначена
    #define Button_h // визначаємо константу Button_h і опис
    // бібліотеки
    #include "Arduino.h"
    // клас обробки  кнопок
    class Button {
      public:
        Button(byte pin, byte timeButton);  // конструктор
        boolean flagPress;
      // ознака натиснення кнопки (сигнал у низькому рівні)
        boolean flagClick;  // ознака кліка
        void scanState();  // метод перевірки стабільного стану
        void setPinTime(byte pin, byte timeButton);
      // встановлення номеру виводу і часу фільтрації
      private:
        byte  _buttonCount;    // лічільник часу фільтрації
        byte _timeButton;      // час фільтрації
        byte _pin;             // номер виводу
    };
    #endif
```

Додаток 2. Файл Button.cpp

```
    #include "Arduino.h"
    #include "Button.h"
    // метод перевірки стану кнопки
    // flagPress= true  - натиснута
    // flagPress= false - не натиснута
    // flagClick= true - була натиснута (клік)
    void Button::scanState() {
     if ( flagPress == (! digitalRead(_pin)) )
         {  // стан сигналу лишався попереднім
            _buttonCount= 0;  // скид лічильника станів сигналу
         }
    else { // стан сигналу змінився
         _buttonCount++;   // +1 до лічильника стану сигналу
         if ( _buttonCount >= _timeButton )
          {// стан сигналу не змінився за заданий час
           // стан сигналу став стійким
```

```
          flagPress= ! flagPress; // інверсія ознаки стану
          _buttonCount= 0;  // скид лічильника стану сигналу
           if ( flagPress == true )
           {flagClick= true;} // ознака кліка
        }
      }
    }
    // метод установки номера виводу і часу підтвердження
    void Button::setPinTime(byte pin, byte timeButton)  {
      _pin= pin;
      _timeButton= timeButton;
      pinMode(_pin, INPUT_PULLUP);  // визначаємо вивід як вхід
    }
    // опис конструктора класу Button
Button::Button(byte pin, byte timeButton) {
      _pin= pin;
      _timeButton= timeButton;
      pinMode(_pin, INPUT_PULLUP);  /// визначаємо вивід як вхід
    }
```

Додаток 3. Бібліотека MSTimer2 Файл MSTimer2.h

```
#ifndef MsTimer2_h
#define MsTimer2_h

#include <avr/interrupt.h>

namespace MsTimer2 {
      extern unsigned long msecs;
      extern void (*func)();
      extern volatile unsigned long count;
      extern volatile char overflowing;
      extern volatile unsigned int tcnt2;

      void set(unsigned long ms, void (*f)());
      void start();
      void stop();
      void _overflow();
}

#endif
```

Додаток 4. Бібліотека MSTimer2 Файл MSTimer2.cpp

```
/*
  MsTimer2.h - Using timer2 with 1ms resolution
*/

#include <MsTimer2.h>

unsigned long MsTimer2::msecs;
void (*MsTimer2::func)();
volatile unsigned long MsTimer2::count;
volatile char MsTimer2::overflowing;
volatile unsigned int MsTimer2::tcnt2;
```

```
	void MsTimer2::set(unsigned long ms, void (*f)()) {
		float prescaler = 0.0;

	#if defined (__AVR_ATmega168__) || defined (__AVR_ATmega48__) ||
defined (__AVR_ATmega88__) || defined (__AVR_ATmega328P__) ||
(__AVR_ATmega1280__)
		TIMSK2 &= ~(1<<TOIE2);
		TCCR2A &= ~((1<<WGM21) | (1<<WGM20));
		TCCR2B &= ~(1<<WGM22);
		ASSR &= ~(1<<AS2);
		TIMSK2 &= ~(1<<OCIE2A);

		if ((F_CPU >= 1000000UL) && (F_CPU <= 16000000UL)) {	//
prescaler set to 64
			TCCR2B |= (1<<CS22);
			TCCR2B &= ~((1<<CS21) | (1<<CS20));
			prescaler = 64.0;
		} else if (F_CPU < 1000000UL) {	// prescaler set to 8
			TCCR2B |= (1<<CS21);
			TCCR2B &= ~((1<<CS22) | (1<<CS20));
			prescaler = 8.0;
		} else { // F_CPU > 16Mhz, prescaler set to 128
			TCCR2B |= ((1<<CS22) | (1<<CS20));
			TCCR2B &= ~(1<<CS21);
			prescaler = 128.0;
		}
	#elif defined (__AVR_ATmega8__)
		TIMSK &= ~(1<<TOIE2);
		TCCR2 &= ~((1<<WGM21) | (1<<WGM20));
		TIMSK &= ~(1<<OCIE2);
		ASSR &= ~(1<<AS2);

		if ((F_CPU >= 1000000UL) && (F_CPU <= 16000000UL)) {	//
prescaler set to 64
			TCCR2 |= (1<<CS22);
			TCCR2 &= ~((1<<CS21) | (1<<CS20));
			prescaler = 64.0;
		} else if (F_CPU < 1000000UL) {	// prescaler set to 8
			TCCR2 |= (1<<CS21);
			TCCR2 &= ~((1<<CS22) | (1<<CS20));
			prescaler = 8.0;
		} else { // F_CPU > 16Mhz, prescaler set to 128
			TCCR2 |= ((1<<CS22) && (1<<CS20));
			TCCR2 &= ~(1<<CS21);
			prescaler = 128.0;
		}
	#elif defined (__AVR_ATmega128__)
		TIMSK &= ~(1<<TOIE2);
		TCCR2 &= ~((1<<WGM21) | (1<<WGM20));
		TIMSK &= ~(1<<OCIE2);

		if ((F_CPU >= 1000000UL) && (F_CPU <= 16000000UL)) {	//
prescaler set to 64
```

```
                TCCR2 |= ((1<<CS21) | (1<<CS20));
                TCCR2 &= ~(1<<CS22);
                prescaler = 64.0;
          } else if (F_CPU < 1000000UL) {   // prescaler set to 8
                TCCR2 |= (1<<CS21);
                TCCR2 &= ~((1<<CS22) | (1<<CS20));
                prescaler = 8.0;
          } else { // F_CPU > 16Mhz, prescaler set to 256
                TCCR2 |= (1<<CS22);
                TCCR2 &= ~((1<<CS21) | (1<<CS20));
                prescaler = 256.0;
          }
     #endif

          tcnt2 = 256 - (int)((float)F_CPU * 0.001 / prescaler);

          if (ms == 0)
                msecs = 1;
          else
                msecs = ms;

          func = f;
     }

     void MsTimer2::start() {
          count = 0;
          overflowing = 0;
     #if defined (__AVR_ATmega168__) || defined (__AVR_ATmega48__) ||
defined (__AVR_ATmega88__) || defined (__AVR_ATmega328P__) ||
(__AVR_ATmega1280__)
          TCNT2 = tcnt2;
          TIMSK2 |= (1<<TOIE2);
     #elif defined (__AVR_ATmega128__)
          TCNT2 = tcnt2;
          TIMSK |= (1<<TOIE2);
     #elif defined (__AVR_ATmega8__)
          TCNT2 = tcnt2;
          TIMSK |= (1<<TOIE2);
     #endif
     }

     void MsTimer2::stop() {
     #if defined (__AVR_ATmega168__) || defined (__AVR_ATmega48__) ||
defined (__AVR_ATmega88__) || defined (__AVR_ATmega328P__) ||
(__AVR_ATmega1280__)
          TIMSK2 &= ~(1<<TOIE2);
     #elif defined (__AVR_ATmega128__)
          TIMSK &= ~(1<<TOIE2);
     #elif defined (__AVR_ATmega8__)
          TIMSK &= ~(1<<TOIE2);
     #endif
     }
```

```
	void MsTimer2::_overflow() {
		count += 1;

		if (count >= msecs && !overflowing) {
			overflowing = 1;
			count = 0;
			(*func)();
			overflowing = 0;
		}
	}

	ISR(TIMER2_OVF_vect) {
	#if defined (__AVR_ATmega168__) || defined (__AVR_ATmega48__) ||
defined (__AVR_ATmega88__) || defined (__AVR_ATmega328P__) ||
(__AVR_ATmega1280__)
		TCNT2 = MsTimer2::tcnt2;
	#elif defined (__AVR_ATmega128__)
		TCNT2 = MsTimer2::tcnt2;
	#elif defined (__AVR_ATmega8__)
		TCNT2 = MsTimer2::tcnt2;
	#endif
		MsTimer2::_overflow();
	}
```

Зміст