ГОСУДАРСТВЕННОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Электропривод и АПУ»

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Методические рекомендации к курсовому проектированию для студентов специальности «Электроэнергетика и электротехника»

Могилев, 2018

УДК 621.3
ББК 31.2
Я 40

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Электропривод и АПУ» «6» февраля 2017 г., протокол № 7

Составитель ст. преподаватель В. Т. Вишнеревский
Рецензент канд. техн. наук, доц. И. В. Лесковец

Методические рекомендации к курсовой работе по дисциплине «Языки программирования» для студентов специальности 13.03.02 «Электроэнергетика и электротехника» по профилю «Электрооборудование автомобилей и тракторов».

Учебно-методическое издание

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Ответственный за выпуск Г. С. Леневский
Технический редактор В. Т. Вишнеревский
Компьютерная верстка В. Т. Вишнеревский

Подписано в печать. 12.01.2018. Формат 60x84/16. Бумага офсетная. Гарнитура Таймс. Печать трафаретная. Усл. печ. л. 2,09. Уч.-изд. л. 2,3. Тираж 50 экз. Заказ № 2391.

Издатель и полиграфическое исполнение:
Государственное учреждение высшего профессионального образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий.
№1/156 от 24.01.2014.
Пр. Мира, 43, 212000, Могилев,

Содержание

Введение

Целью учебной дисциплины «Языки программирования» является подготовка специалистов, умеющих обоснованно применять языки программирования для решения научно-технических задач.

Целью курсовой работы является закрепление на практике знаний, полученных при изучении теоретического курса и выполнении лабораторных работ.

Курсовая работа по дисциплине «Языки программирования» является важным этапом процесса обучения специалистов по профилю подготовки «Электрооборудование автомобилей и тракторов». Электронные системы современных автомобилей и тракторов как правило имеют программное управление и, следовательно, разработка таких систем требует наличия у специалистов навыков программирования. Специфика данных систем такова, что в них, как правило, используются микроконтроллеры, программирование которых отличается рядом специфических особенностей по сравнению с программированием на персональных компьютерах.

В данной курсовой работе студентам предлагается разработать собственное устройство с использованием платы «Arduino», а также подключаемых модулей. Следует обратить внимание на то, что ДЛЯ УСПЕШНОЙ ЗАЩИТЫ КУРСОВОЙ РАБОТЫ **НЕ** ТРЕБУЕТСЯ СОБИРАТЬ РЕАЛЬНОЕ УСТРОЙСТВО, достаточно выполнить функциональную схему, алгоритм, а также управляющую программу и предоставить пояснительную записку с описанием разработанного устройства.

Платформа «Arduino» используется в данной курсовой работе потому, что создание устройств и написание программ в данном случае не требует от студентов специфических знаний и навыков в области электроники, микропроцессорной техники и программирования. Вся требуемая для выполнения работы информация доступна в сети.

Выполнение данной курсовой работы позволяет студенту познакомиться с особенностями программирования микропроцессорных устройств, а также с возможностями платформы «Arduino», что является первыми шагами в изучении микропроцессорных систем управления.

1. Платформа «Arduino»

1.1 Общие сведения о платформе «Arduino»

Arduino - торговая марка аппаратно-программных средств для построения простых систем автоматики и робототехники, ориентированная на непрофессиональных пользователей. **Программная** часть состоит из бесплатной программной оболочки (IDE) для написания программ, их компиляции и программирования аппаратуры. **Аппаратная** часть представляет собой набор смонтированных печатных плат, продающихся как официальным производителем, так и сторонними производителями. Полностью открытая архитектура системы позволяет свободно копировать или дополнять линейку продукции Arduino.

Arduino может использоваться как для создания автономных объектов автоматики, так и подключаться к программному обеспечению на компьютере через стандартные проводные и беспроводные интерфейсы.

Под торговой маркой Arduino выпускается несколько плат с микроконтроллером (*boards*) и платы расширения (*shields*). Большинство плат с микроконтроллером снабжены минимально необходимым набором внешних устройств для нормальной работы микроконтроллера (стабилизатор питания, кварцевый резонатор, цепочки сброса и т. п.).

Arduino и Arduino-совместимые платы спроектированы таким образом, чтобы их можно было при необходимости расширять, добавляя в устройство новые компоненты. Эти платы расширений подключаются к Arduino посредством установленных на них штыревых разъёмов. Существует ряд плат с унифицированным конструктивом, допускающим конструктивно жесткое соединение процессорной платы и плат расширения в стопку через штыревые линейки. Кроме того, выпускаются платы уменьшенных габаритов (например, Nano, Lilypad) и специальных конструктивов для задач робототехники. Независимыми производителями также выпускается большая гамма всевозможных датчиков и исполнительных устройств, в той или иной степени совместимых с базовым конструктивом Ардуино.

В концепцию Arduino не входит корпусной или монтажный конструктив. Разработчик выбирает метод установки и механической защиты плат самостоятельно. Сторонними производителями выпускаются наборы робототехнической электромеханики, ориентированной на работу совместно с платами Arduino.

Общий вид платы Arduino Uno представлен на рисунке 1.

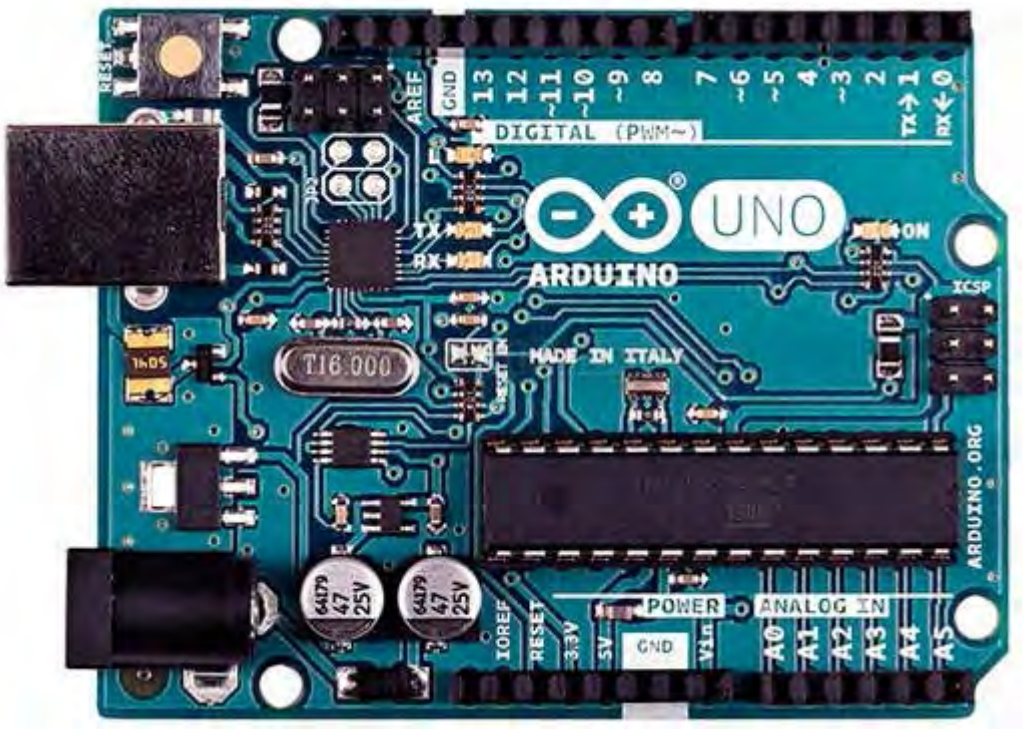


Рисунок 1 - Общий вид платы Arduino Uno

1.2 Датчики, подключаемые к платам «Arduino»

Сторонние производители выпускают широкую гамму датчиков, подключаемых к Arduino. Например, гироскопы, компасы, манометры, гигрометры, термометры.

Для примера рассмотрим ультразвуковой датчик расстояния:

Ультразвуковой датчик расстояния - модуль HC-SR04 использует акустическое излучение для определения расстояния до объекта. Этот бесконтактный датчик обеспечивает высокую точность и стабильность измерений. Диапазон измерений составляет: от 2 см до 400 см. На показания датчика практически не влияют солнечное излучение и электромагнитные шумы. Модуль поставляется в комплекте с трансмиттером и ресивером.

Технические характеристики HC-SR04

- Напряжение питания: +5В – постоянный ток;
- Сила тока покоя: < 2 мА;
- Рабочая сила тока: 15 мА;
- Эффективный рабочий угол: < 15°;
- Расстояние измерений: от 2 см до 400 см (1 – 13 дюймов);
- Разрешающая способность: 0.3 см;
- Угол измерений: 30 градусов;
- Ширина импульса триггера: 10 микросекунд;
- Размеры: 45 мм х 20 мм х 15 мм.

Общий вид датчика расстояния представлен на рисунке 2

Рисунок 2 - Общий вид датчика расстояния

Назначение выводов датчика:

- VCC: +5 вольт (постоянный ток)
- Trig : Триггер (INPUT)
- Echo: Эхо (OUTPUT)
- GND: Земля

Схема подключения датчика расстояния представлена на рисунке 3

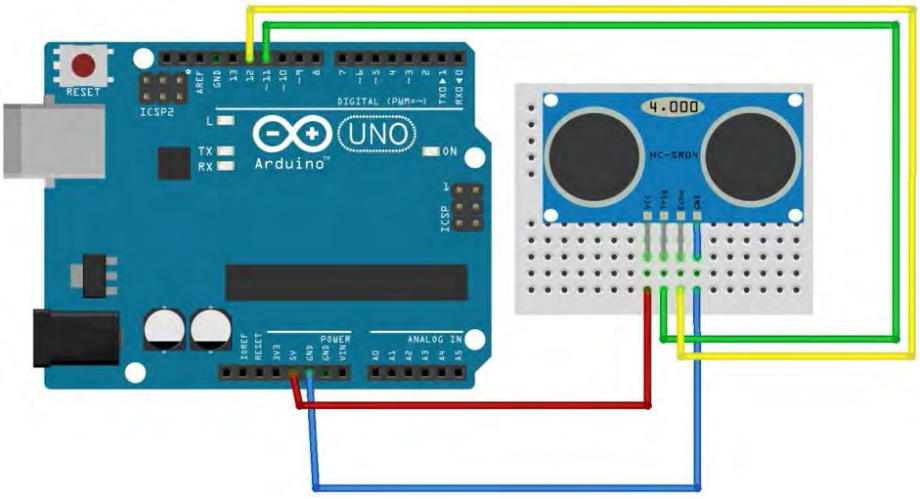


Рисунок 3 - Схема подключения датчика расстояния к Arduino

В данном примере ультразвуковой датчик HC–SR04 определяет расстояние и выводит полученные значения в окно серийного монитора в среде Arduino IDE.

В свободном доступе существует библиотека NewPing, которая облегчает использование HC–SR04.

Ниже представлен пример программы с использованием библиотеки NewPing.

```
#include <NewPing.h> // Подключаем библиотеку
#define TRIGGER_PIN 12 // Назначаем вывод «Триггер»
#define ECHO_PIN 11 // Назначаем вывод «Echo»
#define MAX_DISTANCE 200 // Устанавливаем максимальное расстояние
NewPing sonar(TRIGGER_PIN, ECHO_PIN, MAX_DISTANCE);
// ^Настройка пинов и максимального расстояния^.
void setup() {
Serial.begin(9600); // Настройка последовательного порта
}
void loop() { //Начало бесконечного цикла
delay(50); // Выдержка времени
unsigned int uS = sonar.ping_cm(); // Получаем показания датчика
Serial.print(uS); // Отправляем полученный результат в порт
Serial.println("cm"); // Отправляем в порт "cm"
}
```

Рассмотрим подключение цифрового компаса к Arduino. Микросхема HMC5883L представляет собой 3-х осевой цифровой компас, работающий по шине I^2C. В качестве сенсоров используется три магниторезистивных датчика. Разработчик: компания Honeywell. Напряжение питания составляет 2.2-3.6В. Чувствительность датчика составляет 5 миллигаусс. Схема подключения датчика представлена на рисунке 4.

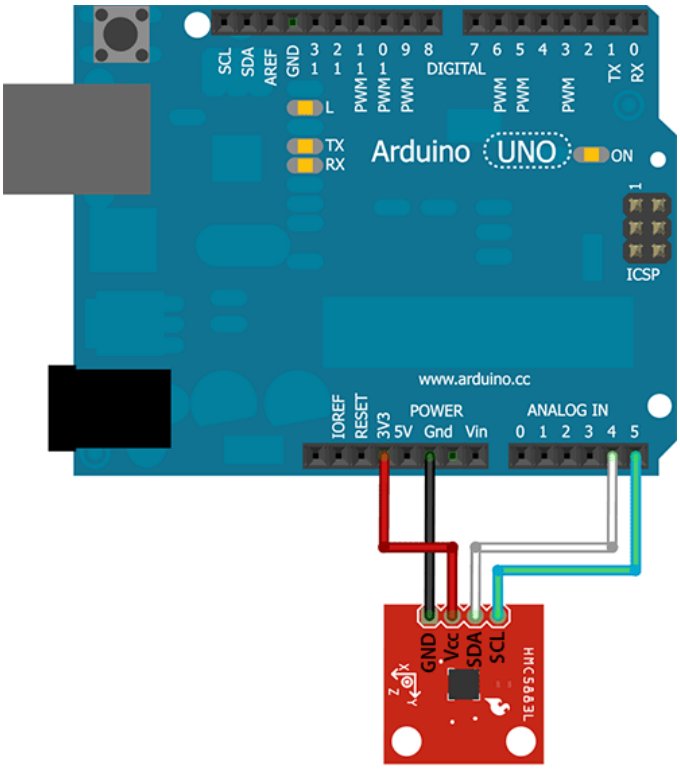


Рисунок 4 — Схема подключения цифрового компаса к Arduino

Датчик может использоваться в мобильных телефонах, планшетах, навигационном оборудовании и прочей потребительской электронике, но для радиолюбителей он может быть интересен тем, что цифровой компас может очень пригодится при конструировании роботов и радиоуправляемых моделей. На странице https://github.com/bildr-org находится готовая библиотека для работы с датчиком. Ниже приведен пример программы:

```
#include "Wire.h"
#include "HMC5883L.h"
HMC5883L compass;
void setup(){
  Serial.begin(9600);
  Wire.begin();
   compass = HMC5883L();  // создаем экземпляр HMC5883L библиотеки
  setupHMC5883L();        // инициализация HMC5883L
}
void loop(){
  float heading = getHeading();
  Serial.println(heading);
  delay(250);
}
void setupHMC5883L(){
  // инициализация HMC5883L, и проверка наличия ошибок
  int error;
  error = compass.SetScale(0.88); // чувствительность датчика из диапазона: 0.88, 1.3, 1.9, 2.5, 4.0, 4.7, 5.6, 8.1
  if(error != 0) Serial.println(compass.GetErrorText(error)); // если ошибка, то выводим ее
  error = compass.SetMeasurementMode(Measurement_Continuous); // установка режима измерений как Continuous (продолжительный)
  if(error != 0) Serial.println(compass.GetErrorText(error)); // если ошибка, то выводим ее
}
float getHeading(){
  // считываем данные с HMC5883L и рассчитываем  направление
  MagnetometerScaled scaled = compass.ReadScaledAxis(); // получаем масштабированные элементы с датчика
  float heading = atan2(scaled.YAxis, scaled.XAxis);    // высчитываем направление
  // корректируем значения с учетом знаков
  if(heading < 0) heading += 2*PI;
  if(heading > 2*PI) heading -= 2*PI;
  return heading * RAD_TO_DEG; // переводим радианы в градусы
}
```

1.3 Исполнительные устройства

Для платформы Arduino предлагаются следующие виды исполнительных устройств:

- модули реле;
- электродвигатели постоянного тока;
- шаговые электродвигатели;
- сервоприводы.

Для примера рассмотрим сервопривод.

Сервопривод - это электропривод, в состав которого входит электродвигатель, механическое передающее устройство, датчик положения и замкнутая система управления. Сервопривод устанавливает свой вал в положение, заданное извне через управляющий вывод и удерживает вал в данном положении.

Для подключения сервопривода используются 3 вывода: земля, питание и управление. Здесь следует обратить внимание на то, что сервопривод может потреблять ток более 1 А во время перемещения, поэтому его следует подключать к отдельному источнику питания.

Общий вид сервопривода представлен на рисунке 5.

Схема подключения сервопривода представлена на рисунке 6.

Рисунок 5 - Общий вид сервопривода

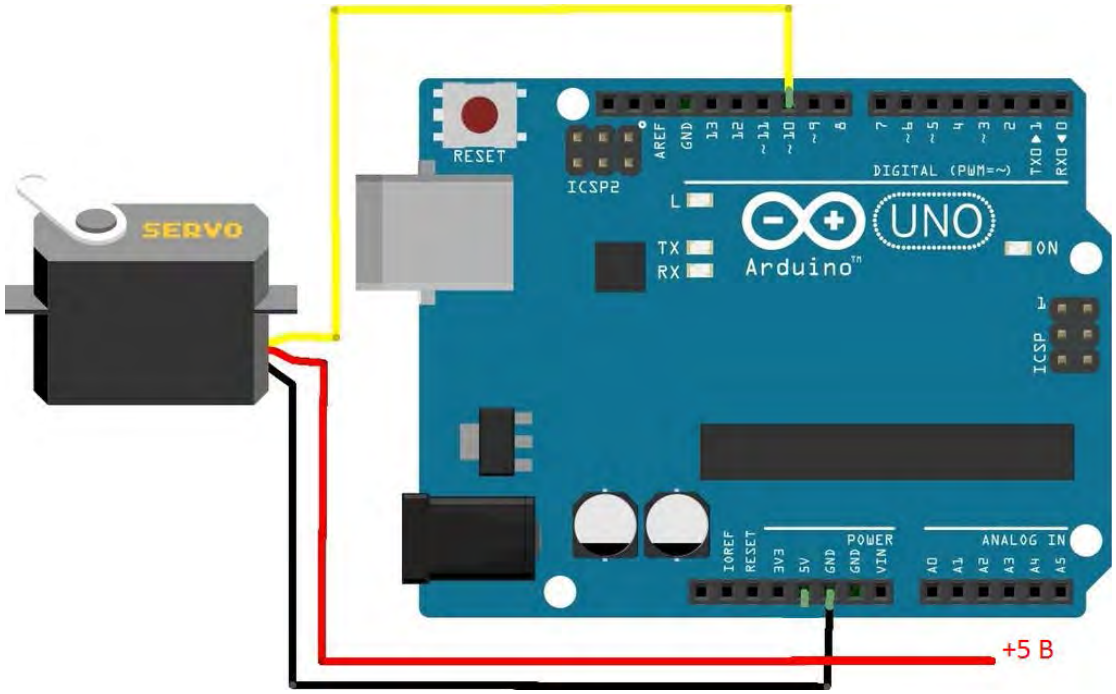


Рисунок 6 - Схема подключения сервопривода к плате Arduino

Пример программы для управления сервоприводом:

```
#include <Servo.h> //используем библиотеку для работы с сервоприводом
Servo servo; //объявляем переменную servo типа Servo
void setup() //процедура setup
{
servo.attach(10); //привязываем привод к порту 10
}
void loop() //процедура loop
{
servo.write(0); //ставим вал под 0
delay(2000); //ждем 2 секунды
servo.write(180); //ставим вал под 180
delay(2000); //ждем 2 секунды
}
```

1.4 Примеры модулей, подключаемых к платам «Arduino»

Одной из особенностей платформы Arduino является возможность использования плат расширения. Плата расширения представляет собой небольшую плату, по размерам, как правило, совпадающую с базовой. Для их использования на большинстве плат Arduino устанавливаются разъемы типа PLD. Стандартизированная интерфейсная часть Arduino позволяет легко устанавливать и производить замену плат расширения. При необходимости возможно использование нескольких плат расширения одновременно. Большое количество различных вариантов исполнения таких плат обеспечивает платформе дополнительную популярность.

В настоящее время существует большое количество плат расширения различного назначения. Самым простым из них является ProtoShield, представляющий собой простую макетную плату. К наиболее распространенным относятся EthernetShield, используемый для передачи информации через сеть Ethernet и MotorShield предназначенный для управления двигателями. Также существуют разнообразные платы для беспроводных модулей типа XBEE, Bluetooth и другие.

В качестве примера платы расширения рассмотрим модуль GPRS (GPRS shield).

GPRS Shield — один из самых простых способов взаимодействовать со своим устройством, даже если оно находится на довольно большом расстоянии. Для взаимодействия требуется наличие сотовой связи.

С этой платой расширения Arduino-устройство получает возможности мобильного телефона. На практике можно использовать данные модули для того, чтобы:

- отправлять через SMS команды на устройство;
- запрашивать состояние или получать уведомления от него через SMS;
- звонить на устройство и проворачивать на нём IVR-сценарии;
- принимать звонки с устройства, чтобы послушать, что происходит вокруг;
- выходить в интернет, опрашивать веб-сервисы, протоколировать данные.

Пример программы с использованием GPRS модуля:

```
// библиотека для работы с GPRS устройством
#include <GPRS_Shield_Arduino.h>
// библиотека для эмуляции Serial порта
// она нужна для работы библиотеки GPRS_Shield_Arduino
#include <SoftwareSerial.h>
// длина сообщения
#define MESSAGE_LENGTH 160
// текст сообщения о включении розетки
```

```
#define MESSAGE_ON  "On"
// текст сообщения о выключении розетки
#define MESSAGE_OFF  "Off"
// текст сообщения о состоянии розетки
#define MESSAGE_STATE  "State"
// пин, к которому подключено реле
#define RELAY 5
// текст сообщения
char message[MESSAGE_LENGTH];
// номер, с которого пришло сообщение
char phone[16];
// дата отправки сообщения
char datetime[24];
bool stateRelay = false;
// создаём объект класса GPRS и передаём в него объект Serial1
GPRS gprs(Serial1);
void setup(){
  // настраиваем пин реле в режим выхода,
  pinMode(RELAY, OUTPUT);
  // подаём на пин реле «низкий уровень» (размыкаем реле)
  digitalWrite(RELAY, LOW);
  // открываем последовательный порт для мониторинга действий в программе
  Serial.begin(9600);
  // включаем GPRS шилд
  gprs.powerOn();
  // ждём, пока не откроется монитор последовательного порта
  // для того, чтобы отследить все события в программе
  while (!Serial) {
  }
  // открываем Serial-соединение с GPRS Shield
  Serial1.begin(9600);
  // проверяем есть ли связь с GPRS устройством
  while (!gprs.init()) {
    // если связи нет, ждём 1 секунду
    // и выводим сообщение об ошибке
    // процесс повторяется в цикле
    // пока не появится ответ от GPRS устройства
    delay(1000);
    Serial.print("GPRS Init error\r\n");
  }
  // вывод об удачной инициализации GPRS Shield
  Serial.println("GPRS init success");}
 void loop(){
  // если пришло новое сообщение
  if (gprs.ifSMSNow()) {
```

```
    // читаем его
    // если есть хотя бы одно непрочитанное сообщение,
    // читаем его
    gprs.readSMS(message, phone, datetime);

    // выводим номер, с которого пришло смс
    Serial.print("From number: ");
    Serial.println(phone);

    // выводим дату, когда пришло смс
    Serial.print("Datetime: ");
    Serial.println(datetime);

    // выводим текст сообщения
    Serial.print("Recieved Message: ");
    Serial.println(message);
    // вызываем функцию изменения состояния реле
    // в зависимости от текста сообщения
    setRelay(phone, message);
  }}
void setRelay(char f_phone[], char f_message[]){
 if (strcmp(f_message, MESSAGE_ON) == 0) {
    // если сообщение — с текстом «On»,
    // выводим сообщение в Serial
    // и подаём на замыкаем реле
    Serial.println("OK! Power is On");
    digitalWrite(RELAY, HIGH);
    stateRelay = true;
    // на номер, с которого пришёл запрос,
    // отправляем смс с текстом о включении питания
    gprs.sendSMS(f_phone, "Power is On");
  } else if (strcmp(f_message, MESSAGE_OFF) == 0) {
    // если пришло сообщение с текстом «Off»,
    // выводим сообщение в Serial
    // и размыкаем реле
    Serial.println("OK! Power is Off");
    digitalWrite(RELAY, LOW);
    stateRelay = false;
    // на номер, с которого пришёл запрос
    // отправляем смс с текстом о выключении питания
    gprs.sendSMS(f_phone, "Power is Off");
  } else if (strcmp(f_message, MESSAGE_STATE) == 0) {
    // если пришло сообщение с текстом «State»,
    // отправляем сообщение с состоянием реле
    if (stateRelay) {
```

```
    Serial.println("State: Power is On");
    gprs.sendSMS(f_phone, "Power is On");
   } else {
    Serial.println("State: Power is Off");
    gprs.sendSMS(f_phone, "Power is Off");
   }
  } else {
   // если сообщение содержит неизвестный текст,
   // отправляем сообщение с текстом об ошибке
   Serial.println("Error... unknown command!");
   gprs.sendSMS(f_phone, "Error...unknown command!");
  }}
```

Рассмотрим подключение платы управления моторами (Motor Shield) к Arduino.

Шаговые двигатели. Шаговые моторы - отличный выбор для управления различными механизмами. Они отлично подходят для проектов с манипуляторами, ЧПУ станками и т.п. на Arduino. Motor Shield поддерживает работу двух шаговых двигателей одновременно. Библиотека работает одинаково для биполярных и униполярных моторов.

Для униполярных моторов: перед подключением моторов надо выяснить как соотносятся контакты и обмотки. Если у мотора пять проводов, будет один, который будет расположен по центру обеих обмоток. Центральный контакт должен быть подключен к контакту GND на плате Motor Shield. Потом надо подключить катушку 1 к одному из контактов для моторов (скажем, M1 или M3), а катушка 2 должна подключаться ко второму порту мотора (M2 или M4).

Для биполярных двигателей: то же самое, что и для униполярных, только нет пятого проводника, который подключен к земле. Скетч такой же.

Последовательность действий при написании программы управления шаговым двигателем следующая:

1. Убедитесь, что вы подключили библиотеку #include <AFMotor.h>

2. Создайте объект для шагового двигателя: AF_Stepper(steps, stepper#). Steps содержит информацию о том, сколько шагов на один оборот совершает мотор. В 7.5 degree/step моторе 360/7.5 = 48 steps. Stepper# отвечает за порт, к которому подключен шаговый двигатель. Если вы используете M1 и M2, это порт 1. Если вы используете M3 и M4 - это порт 2.

3. Установите скорость мотора, используя setSpeed(rpm), rpm - задаваемое количество оборотов в минуту на шаговом двигателе.

4. После этого каждый раз, когда вы хотите обеспечить вращение ротора мотора, надо вызывать процедуру step(#steps, direction, steptype).#steps - это то,

сколько шагов вы хотите совершить. direction - это FORWARD или BACKWARD, а тип шага - это SINGLE, DOUBLE. INTERLEAVE или MICROSTEP.

"Single" означает активацию одной катушки, "double" означает одновременную активацию 2-х катушек (для большего крутящего момента), а 2 "interleave" - среднее между режимами 'single' и 'double' (в результате скорость будет в два раза меньше). "Microstepping" - метод, с помощью которого обеспечивается плавное перемещение между шагами. Вы можете использовать любой тип шага, менять методы "на лету" в зависимости от того, что вам необходимо - минимальная мощность, больший крутящий момент или большая точность.

5. По умолчанию мотор будет удерживать положение после того, как отработает заданное количество шагов. Если вы хотите отключить катушки, чтобы обеспечить свободное вращение вала двигателя, можете использовать функцию release().

6. Команды шагов блокируются и становятся доступными только после отработки заданных шагов.

Так как как команды шагов блокируются - надо подавать новые команды каждый раз, когда вы хотите обеспечить вращение ротора. Если вы хотите расширить возможности шаговых приводов, попробуйте загрузить библиотеку AccelStepper library (устанавливается так же как и библиотека AFMotor), в которой есть интересные примеры управления двумя шаговыми двигателями одновременно с разными ускорениями.

Ниже представлен пример кода для управления шаговым двигателем:

```
#include <AFMotor.h>
AF_Stepper motor(48, 2);
void setup() {
Serial.begin(9600); // устанавливаем скорость обмена данными на 9600 bps
Serial.println("Stepper test!");
motor.setSpeed(10); // 10 rpm
motor.step(100, FORWARD, SINGLE);
motor.release();
delay(1000);
}
void loop() {
motor.step(100, FORWARD, SINGLE);
```

```
motor.step(100, BACKWARD, SINGLE);
motor.step(100, FORWARD, DOUBLE);
motor.step(100, BACKWARD, DOUBLE);
motor.step(100, FORWARD, INTERLEAVE);
motor.step(100, BACKWARD, INTERLEAVE);
motor.step(100, FORWARD, MICROSTEP);
motor.step(100, BACKWARD, MICROSTEP);
}
```

Двигатели постоянного тока (DC motors) используются во многих робототехнических проектах.

Motor Shield дает возможность управлять одновременно 4-ми двигателями постоянного тока, обеспечивая вращение роторов в двух направлениях. То есть, вал может вращаться и по часовой стрелке и против. Можно управлять скоростью вращения с инкрементом 0.5 %.

Не забывайте, что H-bridge чип не рассчитан на нагрузки больше 0.6 А при пиковых нагрузках 1.2 А. То есть, шилд рассчитан на маленькие моторы.

Для того, что бы подключить мотор, просто установите два контакта в терминалы, к контактам M1, M2, M3 или M4. После этого используйте скетч, пояснения к которому приведены ниже.

1. Убедитесь, что вы подключили библиотеку #include <AFMotor.h>

2. Создайте объект AF_DCMotor с такими свойствами: AF_DCMotor(motor#, frequency). У созданного объекта два аргумента.

Первый отвечает за порт, к которому подключен мотор 1, 2, 3 или 4.

frequency отвечает за частоту управляющего сигнала.

Для моторов 1 и 2 вы можете выбрать MOTOR12_64KHZ, MOTOR12_8KHZ, MOTOR12_2KHZ, или MOTOR12_1KHZ.

3. После этого вы можете настроить скорость мотора с использованием setSpeed(speed), где speed задается в диапазоне от 0 (остановка) до 255 (максимальная скорость).

4. Для ого, чтобы запустить мотор, необходимо вызвать функцию run(direction), где direction - это FORWARD, BACKWARD или RELEASE. Естественно, Arduino не знает, что такое "вперед", или "назад". Так что "вперед" - "назад" вы должны настроить на собственное усмотрение, поменяв контакты "+"/"-" от привода к клемам мотор шилда.

Ниже приведен пример программы для управления электродвигателем постоянного тока.

```
#include <AFMotor.h>
AF_DCMotor motor(2, MOTOR12_64KHZ); // создаем объект motor #2, 64 КГц ШИМ
void setup() {
Serial.begin(9600); // Устанавливаем скорость передачи данных 9600 bps
Serial.println("Motor test!");
motor.setSpeed(200); // устанавливаем скорость на 200 из 255 допустимых
}
void loop() {
Serial.print("tick");
motor.run(FORWARD); // ротор двигателя начинает вращаться "вперед"
delay(1000);
Serial.print("tock");
motor.run(BACKWARD); // вращается в противоположном направлении
delay(1000);
Serial.print("tack");
motor.run(RELEASE); // остановка
delay(1000);
}
```

2. Составление алгоритмов

Алгоритм – это инструкция о том, в какой последовательности нужно выполнить действия при переработке исходного материала в требуемый результат. [последовательность точных предписаний, понятных исполните лю (компьютеру, роботу и пр.), совершить последо вательность действий, направленных на достиже ние конкретного результата.]

Наряду с понятием алгоритма используют термин *алгоритмизация* , под которой понимают совокупность приемов и способов составления алгоритмов для решения алгоритмических задач.

Часто алгоритм используется не как инструкция для автомата, а как схема алгоритмического решения задачи. Это позволяет оценить эффективность предлагаемого способа решения, его результативность, исправить возможные ошибки, сравнить его до применения на компьютере с другими алгоритмами решения этой же задачи. Наконец, алгоритм является основой для составления программы, которую пишет программист на каком-либо языке программирования с тем, чтобы реализовать процесс обработки данных на компьютере.

Неотъемлемым свойством алгоритма является его результативность , то есть алгоритмическая инструкция лишь тогда может быть названа алгоритмом, когда при любом сочетании исходных данных она гарантирует, что через конечное число шагов будет обязательно получен результат.

На практике получили известность два способа изображения алгоритмов:

в виде пошагового словесного описания;

в виде блок-схем.

Первый из этих способов получил значительно меньшее распространение из-за его многословности и отсутствия наглядности. Второй, напротив, оказался очень удобным средством изображения алгоритмов и получил широкое распространение в научной и учебной литературе. Именно этот способ следует использовать при составлении и описании алгоритма в данной курсовой работе.

Блок-схема – это последовательность блоков, предписывающих выполнение определенных операций, и связей между этими блоками. Внутри блоков указывается информация об операциях, подлежащих выполнению. Конфигурация и размеры блоков, а также порядок графического оформления блок-схем регламентированы ГОСТ 19002-80 и ГОСТ 19003-80 "Схемы алгоритмов и программ".

В табл. 1 приведены наиболее часто используемые блоки, изображены элементы связей между ними и дано краткое пояснение к ним. Блоки и элементы связей называют элементами блок-схем.

Представленных в таблице элементов вполне достаточно для изображения алгоритмов, которые необходимы при выполнении студенческих работ.

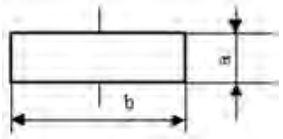
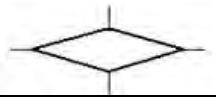
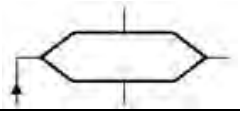
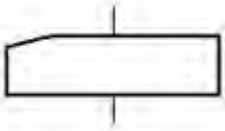
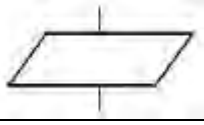
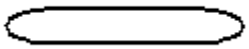
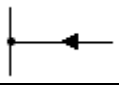
При соединении блоков следует использовать только вертикальные и горизонтальные линии потоков.

Горизонтальные потоки, имеющие направление справа налево, и вертикальные потоки, имеющие направление снизу вверх, должны быть обязательно помечены стрелками.

Прочие потоки могут быть помечены или оставлены непомеченными.

Линии потоков должны быть параллельны линиям внешней рамки или границам листа

Таблица 1

Название	**Элемент**	**Комментарий**
Процесс		Вычислительное действие или последовательность вычислительных действий
Решение		Проверка условия
Модификация		Заголовок цикла
Предопределенный процесс		Обращение к процедуре
Документ		Вывод данных, печать данных
Перфокарта		Ввод данных
Ввод/Вывод		Ввод/Вывод данных
Соединитель		Разрыв линии потока
Начало, Конец		Начало, конец, пуск, останов, вход и выход во вспомогательных алгоритмах
Комментарий		Используется для размещения надписей
Горизонтальные и вертикальные потоки		Линии связей между блоками, направление потоков
Слияние		Слияние линий потоков

Межстраничный соединитель		Нет

Расстояние между параллельными линиями потоков должно быть не менее 3 мм , между остальными элементами схемы – не менее 5 мм .

Горизонтальный и вертикальный размеры блока должны быть кратны 5 мм (делиться на 5 нацело). Отношение горизонтального и вертикального размеров блока b/a = 1.5 является основным. При ручном выполнении блока допустимо отношение b/a = 2.

Блоки "Начало", "Конец" и "Соединитель" имеют высоту a/2, т. е. вдвое меньше основной высоты блоков.

Для размещения блоков рекомендуется поле листа разбивать на горизонтальные и вертикальные (для разветвлявшихся схем) зоны.

Для удобства описания блок-схемы каждый ее блок следует пронумеровать. Удобно использовать сквозную нумерации блоков. Номер блока располагают в разрыве в левой верхней части рамки блока.

По характеру связей между блоками различают алгоритмы линейной, разветвляющейся и циклической структуры.

Примеры, пояснявшие изложенное, можно найти в блок-схемах алгоритмов, которые будут приведены ниже.

3. Написание программ для «Arduino»

3.1 Общие принципы написания программ для «Arduino»

В программе, предназначенной для платы Arduino и разработанной в Arduino IDE, обязательно должны присутствовать 2 функции: **setup()** и **loop()**.

```
void setup() {
// put your setup code here, to run once:
}
void loop() {
// put your main code here, to run repeatedly:
}
```

Функция **setup()** запускается один раз, после каждого включения питания или сброса платы Arduino. В теле данной функции пишется код для инициализации переменных, установки режима работы цифровых портов, и т.д. В дальнейших примерах вы увидите этот механизм.

Функция **loop()** в бесконечном цикле последовательно раз за разом исполняет команды, которые описаны в её теле. Т.е. после завершения функции снова произойдет её вызов.

Пример инициализации цифрового выхода:

```
void setup() {
int led = 13;
```

```
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}
```

Пример реализации мигания светодиодом в функции loop():

```
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

3.2 Использование прерываний в «Arduino»

Прерывание (англ. *interrupt*) — сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается, и управление передаётся обработчику прерывания.

Обработчик прерывания (функция обработки прерывания, процедура обработки прерывания) по английски называется *Interrupt Service Routine*, или сокращенно *ISR,* реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

Вектор прерывания — это указатель на адрес расположения инструкций, которые должны быть выполнены при вызове данного прерывания. То есть это адрес программы обработки данного прерывания. Векторы прерываний объединяются в **таблицу векторов прерываний**, содержащую адреса обработчиков прерываний. Местоположение таблицы зависит от типа и режима работы процессора.

Пример прерывания:

```
const byte LED = 13;
const byte BUTTON = 2;
// Функция обработки прерывания (ISR)
void pinChange ()
{
if (digitalRead (BUTTON) == HIGH)
  digitalWrite (LED, HIGH);
else
  digitalWrite (LED, LOW);
}
```

```
void setup ()
{
  pinMode (LED, OUTPUT);  // устанавливаем пин в режим вывода
  digitalWrite (BUTTON, HIGH);  // встроенный подтягивающий резистор
  attachInterrupt (0, pinChange, CHANGE); // подключаем обработчик прерывания
}
void loop ()
{
  // не делаем здесь ничего
}
```

Этот пример показывает, что, несмотря на то, что в основном цикле *loop* не выполняется никаких действий, вы можете включать и выключать светодиод на 13 пине, нажимая кнопку, подключенную к *D2*.

Чтобы проверить это, просто нажмите кнопку, подключенную между 2 выводом и землей. Внутренний подтягивающий резистор (подключается в функции *setup*) в нормальном состоянии (когда нет внешнего сигнала) поддерживает этот вывод в состоянии *HIGH*. Когда вы замыкаете его на замлю, он переходит в состояние *LOW*. Изменение состояния пина определяется при помощи прерывания *CHANGE*, которое является причиной вызова функции обработки прерывания.

В более сложном примере, основной цикл *loop* может делать что-то делать полезное, например, снимать показания температуры и позволять обработчику прерывания обнаруживать нажатие кнопки.

Полный список прерываний можно найти в документации на микроконтроллер, входящий в состав платы Arduino.

Вывод процессора из спящего режима.

Внешние прерывания, прерывания по изменению состояния пинов и прерывание сторожевого таймера также могут быть использованы для пробуждения процессора. Это может быть очень удобно, так как в спящем режиме процессор может быть настроен на меньшее энергопотребление (около 10 мкА). Прерывания по фронту, спаду или низкому уровню сигнала могут использоваться, чтобы разбудить гаджет. Например, если вы нажмете кнопку на нем, или же можно будить его периодически, используя прерывание сторожевого таймера, чтобы, к примеру, проверить время или температуру.

Прерывания по изменению состояния пинов могут использоваться для пробуждения процессора при нажатии кнопки на клавиатуре или чем-то похожим.

Процессор также может быть выведен из спящего режима прерыванием таймера при достижении определенного значения или переполнения и некоторыми другими событиями, такими как входящее сообщение I^2C.

Разрешение и запрет прерываний

Прерывание *RESET* не может быть запрещено, но другие прерывания могут временно запрещаться сбросом флага прерывания.

Вы можете разрешить прерывания, испольуя функции *interrupts* или *sei*:

```
interrupts(); // или...
sei(); // устанавливаем флаг прерывания
```

Если вам нужно запретить прерывания, можно воспользоваться функцией noInterrupts или сбросить флаг прерывания:

```
noInterrupts(); // или ...
cli(); // сбрасываем флаг прерывания
```

Что такое приоритет прерывания?

Поскольку существует 25 прерываний (за исключением *RESET*), возможно что произойдет одновременно больше, чем одно прерывание или же прерывание произойдет раньше, сем будет обработано другое. Также событие прерывания может произойти в момент, когда прерывания запрещены.

Приоритетный порядок является последовательностью, в которой процессор следит за событиями прерываний. Более высокая позиция в списке означает более высокий приоритет. Так, например, внешнее прерывание 0 (пин *D2*) будет обработано перед внешним прерыванием 1 (пин *D3*).

Может прерывание произойти пока прерывания запрещены?

События прерываний (именно события) могут произойти в любое время и большинство из них запоминается, установкой флага *interrupt event* внутри процессора. Если прерывания запрещены, то это прерывание может быть обработано, когда вновь станет разрешено в порядке приоритетности.

Что такое переменные *volatile*?

Переменные, к которым имеют доступ функции обработки прерываний и обычные функции должны быть объявлены как *volatile*. Такое объявление сообщит компилятору, что эти переменные могут измениться в любой момент и поэтому компилятор должен перезагрузить переменную каждый раз, когда вы ссылаетесь на нее, а не полагаться на копию, которая может храниться в регистре процессора:

```
volatile boolean flag;
// Процедура обработки прерывания (ISR)
void isr ()
{
flag = true;
}
void setup ()
{
attachInterrupt (0, isr, CHANGE);  // подключаем обработчик прерывания
```

```
}
void loop ()
{
  if (flag)
  {
  // произошло прерывание
  }
}
```

Может ли обработчик прерывания быть прерван?

Когда производится вход в обработчик прерывания, то **прерывания запрещаются**. Естественно, что они должны были быть включены в первую очередь, в противном случае в обработчик прерывания не попасть. Однако, чтобы избежать то, что сама процеда обработуи прерывания прерывается, процессор запрещает прерывания.

При выходе из обработчика, прерывания вновь разрешаются. Компилятор также генерирует код внутри обработчика прерываний для сохранения регистроф и статусных флагов для того, что бы все что вы делали до возникновения прерывания не было затронуто.

Тем не менее, вы можете включить прерывания внутри обработчика прерываний, если посчитаете, что вам это необходимо. Например:

```
// Процедура обработки прерывания (ISR)
void pinChange ()
{
  // здесь обработка изменения состояния пина
  interrupts ();  // позволить еще прерывания
}
```

Однако, с этим приемом нужно быть осторожными, так как другое прерывание может вызвать рекурсивный вызов pinChange и, вполне возможно, приведет к нежелательному результату.

Сколько времени проходит до того, как процессор начинает входить в обработчик?

Тут могут быть различные варианты. Приведенные выше цифры являются идеальными для случая, когда прерывание обрабатывалось немедленно. Некоторые факторы могут привнести задержку:

- Если процессор в режиме сна, то существует время пробуждения, которое может занять до нескольких миллисекунд пока тактирование опять наберет скорость. Это время будет зависеть от установок фьюзов (конфигурационных регистров) и того, насколько глубоко процессор ушел в сон.
- Если процедура обработки прерывания уже выполняется, то следующие прерывания не могут быть введены либо до ее окончания, либо пока они

не будут разрешены в самой этой процедуре. Вот почему нужно делать процедуру обработки прерывания как можно короче, каждая микросекунда, которая будет здесь потрачена, затянет выполнение другой обработки.

- Иногда код выключает прерывания. Например, вызов *millis* ненадолго выключает прерывания. Поэтому, время обработки прерывания увеличится на то время, пока прерывания были запрещены.
- Прерывания могут обрабатываться только в конце инструкции процессора. Так, если конкретная команда занимает три такта, и она только началась, то прерывание будет отложено, по крайней мере, на пару тактов.
- Событие, которое возвращает прерывание обратно (например, возвращение из процедуры обработки прерывания) гарантированно выполнит, по крайней мере, еще одну инструкцию. Таким образом, даже если процедура обработки прерывания уже завершилась и ожидает следующее прерывание, то ему придется ожидать не менее одной инструкции, прежде чем оно будет обработано.
- Так как прерывания имеют приоритет, то прерывания с более высоким приоритетом могут быть обработаны до нужного вам прерывания.

Производительность

Прерывания могут повысить производительность во многих ситуациях потому, что позволяют реализовать основную программу без постоянного отслеживания, например, нажатия кнопки. Однако, служебные процедуры, обслуживающие прерывания, как уже сказано выше, будут выполняться дольше, чем циклический опрос входного порта. Если вам совершенно необходимо, отреагировать на события, скажем, за микросекунды, то прерывание для этого будет слишком медленным. В этом случае вы можете отключить прерывания (например, таймеры) и просто следить в цикле за изменением состояния пина.

Советы по написанию функции обработки прерывания

Делайте обработчик прерывания как можно короче! Пока выполняется обработчик прерывания, вы не можете обрабатывать другие прерывания. Таким образом, можно легко пропустить нажатие кнопки или входящее сообщение по *Serial*-порту если сделать его слишком большим. В частности, в обработчик не нужно помещать отладочные операторы *print*. Это, скорее, создаст проблем больше, чем поможет решить.

Разумным будет установить однобайтный флаг, а затем проверять его в основном цикле *loop*. Или сохранить входящий байт из последовательного порта в буфер. Встроенные прерывания таймера отслеживают время, прошедшее с момента возникновения внутреннего переполнения таймера и, таким образом, вы можете работать с прошедшим временем, зная, сколько раз таймер переполняется.

Напомню, что внутри обработчика все прерывания отключены. Таким образом, надеясь, что время, возвращаемое при вызове функции *millis* изменится, вы разочаруетесь. Можно получить время этим способом, но нужно помнить, что таймер не увеличивается. Слишком долго находясь в обработчике, таймер может пропустить событие переполнения, что приведет к тому, что время, возвращаемое *millis* станет неверным.
Тест показал, что для микроконтроллера *ATmega328*, работающего на частоте 16 МГц, вызов функции *micros* занимает 3.5625 мкс, вызов *millis* занимает 1.9375 мкс. Запись (сохранение) текущего значения таймера разумно сделать в обработчике прерывания. Определение числа прошедших миллисекунд быстрее, чем прошедших микросекунд (количество миллисекунд просто извлекается из переменной). Однако количество микросекунд определяется путем добавления текущего значения таймера 0 (хранит увеличение) к сохраненному «*Timer 0 overflow count*».

Прерывания по изменению состояния вывода

Существует два способа определить внешние события на пине. Первый — это специальное «внешнее прерывание» выводов *D2* и *D3*. Это внешние дискретные события прерывания, по-одному на пин. Вы можете получить их, используя *attachInterrupt* для каждого пина. Вы можете определить условия по фронту, спаду, изменению или же низкому уровню для прерывания.
Однако, также существуют прерывания по «изменению пина» для всех выводов (верно для *ATmega328*). Они действуют на группы выводов: *D0-D7*, *D8-D13*, *A0-A5*. Имеют более низкий приоритет, чем события для внешних прерываний. Можно реализовать обработчик прерываний для обработки изменений на пинах *D8-D13* следующим образом:

```
ISR (PCINT0_vect)
{
  // состояние одного из выводов D8-D13 изменилось
}
```

Очевидно, что необходим дополнительный код для определения того, состояние какого вывода/выводов изменились (например, сравнением с предыдущим значением).
Каждое прерывание по изменению состояния пина имеет связанный байт «маски» в процессоре, так что возможно сконфигурировать их реагировать только, например, на *D8*, *D10* и *D12*, а не на изменения любого из *D8-D13*. Однако, по-прежнему нужны дополнительные операции, чтобы выяснить, состояние каких именно выводов изменилось.

Неявное использование прерываний

Окружение *Arduino* уже использует прерывания, даже если вы персонально и не пытались делать это. При вызове функций millis и micros используется свойство «переполнение таймера». Один из внутренних таймеров (timer 0) настроен на прерывание примерно 1000 раз в секунду, и инкрементирует внутренний счетчик, который эффективно становится счетчиком millis. Небольшим улучшением может стать настройка точного значения тактовой частоты.

Также, аппаратная библиотека для передачи данных через *Serial*-соединение использует прерывания для обработки входящих и исходящих данных. Это очень полезно, так как ваша программа может делать что-то еще, пока не произойдет прерывание и заполнит внутренний буфер. Затем, когда производится проверка *Serial.available*, можно выяснить, есть ли что-нибудь в этом буфере.

Выполнение следующей инструкции после разрешения прерываний.
Существует три основных способа разрешить прерывания, которые не были до этого включены:

```
sei ();  // установить флаг разрешения прерываний
SREG |= 0x80;  // установить старший бит в статусном регистре
reti  ;   // использовать ассемблерную инструкцию "return from interrupt"
```

Во всех случаях, процессор гарантирует, что **следующая** инструкция после того, как прерывания разрешены (если они были ранее запрещены) всегда выполнится, даже если событие прерывания находится в ожидании. Под «следующей» инструкцией здесь понимается следующая в программной последовательности, а не обязательно следующая в коде.

Это позволяет написать следующий код:

```
sei ();
sleep_cpu ();
```

Если бы не эта гарантия, прерывание могло бы произойти до того, как процессор ушел в спящий режим, а затем, возможно, он никогда бы не вернулся из данного режима.

4. Примеры заданий для курсовой работы

Возможные варианты тем курсовой работы следующие:

1. Дистанционное управление освещением.
2. Автоматическое управление освещением.
3. Дистанционное управление нагревом.
4. Автоматическое управление поливом растений.
5. Измерение напряжения с выводом на индикатор.
6. Измерение тока с выводом на индикатор.
7. Измерение емкости конденсаторов.
8. Измерение индуктивности катушек.
9. Измерение температуры с сохранением данных.
10. Измерение температуры с оповещением.
11. Система стабилизации температуры.
12. Цифровой тахометр.
13. Измерение ускорения с регистрацией данных.
14. Газоанализатор.
15. Пожарный извещатель.
16. Часы с будильником.
17. Цифровой таймер.
18. Управление жалюзи.
19. Управление отоплением комнаты.
20. Охранная система для дачи.
21. Автомобильный иммобилайзер.
22. Противоугонная система для автомобиля.
23. Дистанционный запуск автомобиля.
24. Мобильный робот с ультразвуковыми датчиками.
25. Мобильный робот с датчиком положения.
26. Робот-манипулятор с сервоприводами.
27. Робот-манипулятор с шаговыми двигателями.
28. Числовое программное управление.
29. Техническое зрение.
30. И т.д.

5. Содержание курсовой работы

5.1 Пояснительная записка

Пояснительная записка к курсовой работе должна содержать следующие разделы:

1. Введение. (Цель работы, описание проблемы)
2. Постановка задачи. (Более подробное описание проблемы, поиск пути ее решения, выбор средств для решения поставленной задачи)
3. Разработка функциональной схемы устройства. (Описание принципа действия и основных взаимосвязей в разрабатываемой системе, описание действий, выполняемых системой)
4. Описание выбранной для решения задачи платы Arduino. (Модель платы, ее параметры, возможности, краткое описание)
5. Описание выбранных датчиков, исполнительных устройств и подключаемых модулей. (Принцип действия, параметры, возможности, взаимодействие с платой Arduino)
6. Разработка алгоритма программы. (Подробное описание последовательности действий, совершаемых программой)
7. Разработка кода программы (Описание кода, описание используемых функций, их параметры и возвращаемые значения)
8. Приложение с кодом разработанной программы.

5.2 Графическая часть

Графическая часть курсовой работы включает в себя схему функциональную и схему алгоритма, которые выполняются на двух отдельных листах формата А1 либо А2, шифр - Д1 (Документы прочие). Функциональная схема изображается в виде произвольных блоков в сочетании с условными изображениями при необходимости.

5.3 Требования к оформлению курсовой работы

Пояснительная записка должна быть оформлена в соответствии с ГОСТ 2.105 — 2003 «Правила оформления тектстовых документов». Схема алгоритма выполняется в соответствии с ГОСТ 19002-80 и ГОСТ 19003-80 "Схемы алгоритмов и программ". Функциональная схема должна быть выполнена аккуратно, ее элементы должны быть четкими. При оформлении на листах больших форматов рекомендуется использовать возможности векторной графики.

6. Типовые замечания, возникающие при проверке курсовых работ

Наиболее часто допускаемые студентами при выполнении курсовой работы ошибки следующие:

1. Неправильно оформленные блоки алгоритма. Обратите особое внимание на блок «Начало, конец» в таблице 1.
2. Отсутствие стрелок на линиях, соединяющих блоки алгоритма.
3. На листе с алгоритмом ставить масштаб не нужно.
4. В правом нижнем углу основной надписи пишется название университета и номер группы.
5. При написании введения студенты часто начинают перечислять комплектующие, которые используются в их проекте. Вместо этого нужно кратко описать что вы собираетесь делать, какие функции будет выполнять ваше устройство и программное обеспечение, чем оно отличается от ближайших аналогов.
6. Часто отсутствуют названия рисунков и таблиц. Название рисунка должно быть написано снизу, а таблицы — сверху. В тексте пояснительной записки должны быть ссылки на все рисунки и таблицы.
7. Поскольку данная курсовая работа является одной из первых, студенты иногда считают, что делать графическую часть необязательно. Это не так.
8. Поскольку при выполнении данной курсовой работы студенты часто пользуются электронными ресурсами, возникают ошибки при оформлении электронного ресурса в вписке литературы.
9. В тех разделах пояснительной записки, которые посвящены разработке алгоритма и кода программы, нужно не просто вставить изображения с фрагментами алгоритма и участки кода программы, но и представить хотя бы краткое их описание. Нужно пояснить, как работает алгоритм, чтобы условно человек, который впервые его видит, мог разобраться. Также нужно представить описание библиотек и функций, которые были использованы при разработке кода программы.

Список рекомендуемых источников

1. Блум, Джереми. Изучаем Arduino: инструменты и методы технического волшебства: Пер. с англ. — СПб.: БХВ-Петербург, 2015. — 336 с.: ил.

2. Петин В.А. Проекты с использованием контроллера Arduino. - 2-е изд., перераб. и доп. - Спб.: БХВ-Петербург, 2015. - 464 с.: ил. - (Электроника)

3. Петин В.А. Arduino и Raspberry Pi в проектах Internet of Things. — СПб.: БХВ-Петербург, 2016. — 320 с.: ил. — (Электроника)

4. Иго Т. Arduino, датчики и сети для связи устройств: Пер. с англ. — 2-е изд. — СПб.: БХВ-Петербург, 2015. — 544 с.: ил.

5. Монк С . Практическая электроника: иллюстрированное руководство для радиолюбителей. М.: Вильямс, 2016.

6. Соммер У. ограммирование микроконтроллерных плат Arduino/Freeduino. — СПб.: БХВ└ Петербург, 2012. — 256 с.: ил. — (Электроника)

7. Момот М. Мобильные роботы на базе Arduino - Спб.: БХВ-Петербург, 2017. - 288 с.: ил.

8. Официальный сайт компании Роботоша [Электронный ресурс] / Прерывания на Arduino — Режим доступа: http://robotosha.ru/arduino/arduino-interrupts.html, свободный. – Загл. с экрана. – Яз. рус., англ.

9. Официальный сайт компании Arduino-diy.com [Электронный ресурс] / Мотор шилд для Arduino — Режим доступа: http://arduino-diy.com/arduino-motor-shield, свободный. – Загл. с экрана. – Яз. рус., англ.

10. Arduino UNO уроки [Электронный ресурс] / Цифровой компас для Arduino - Режим доступа: http://cxem.net/arduino/arduino72.php, свободный. – Загл. с экрана. – Яз. рус., англ.