http://arduino.ua

Функция Setup

setup()

Функция setup() вызывается, когда стартует скетч. Используется для инициализации переменных, определения режимов работы выводов, запуска используемых библиотек и т.д. Функция setup запускает только один раз, после каждой подачи питания или сброса платы Arduino.
Пример

```
int buttonPin = 3;
void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}
void loop()
{
  // ...
}
```

Функция Loop

loop()

После вызова функции setup(), которая инициализирует и устанавливает первоначальные значения, функция loop() делает точь-в-точь то, что означает её название, и крутится в цикле, позволяя вашей программе совершать вычисления и реагировать на них. Используйте её для активного управления платой Arduino.

Пример

```
int buttonPin = 3;
// setup инициализирует последовательный порт и кнопку
void setup(){
  beginSerial(9600);
  pinMode(buttonPin, INPUT);
}
// в цикле проверяется состояние кнопки,
// и на последовательный порт будет отправлено сообщение, если
она нажата
void loop(){
  if (digitalRead(buttonPin) == HIGH)
    serialWrite('H');
  else
    serialWrite('L');

  delay(1000);
}
```

Оператор If

if (условие) и ==, !=, <, > (операторы сравнения)

if, используется в сочетании с операторами сравнения, проверяет, достигнута ли истинность условия, например, превышает ли входное значение заданное число. Формат оператора **if** следующий:

```
if (someVariable > 50)
{
// выполнять действия
}
```

Программа проверяет, значение *someVariable* больше чем 50 или нет. Если да, то выполняются определенные действия. Говоря иначе, если выражение в круглых скобках истинно, выполняются операторы внутри фигурных скобок. Если нет, программа пропускает этот код.

Скобки после оператора **if** могут быть опущены. Если так сделано, только следующая строка (обозначенная точкой с запятой) становится оператором, выполняемым в операторе **if**.

```
if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120)
digitalWrite(LEDpin, HIGH);

if (x > 120){ digitalWrite(LEDpin, HIGH); }

if (x > 120){
digitalWrite(LEDpin1, HIGH);
digitalWrite(LEDpin2, HIGH);
}                                    // все правильно
```

Выражения, которые вычисляются внутри круглых скобок, могут состоять из одного или нескольких операторов.

Операторы сравнения

x == y (x равно y)

x != y (x не равно y)

x < y (x меньше чем y)

x > y (x больше чем y)

x <= y (x меньше чем или равно y)

x >= y (x больше чем или равно y)

Внимание!

Следите, чтобы случайно не использовать знак простого равенства (например, if (x = 10)). Знак простого равенства – это оператор присваивания, и устанавливает значение x равное 10 (заносит значение 10 в переменную x). Вместо этого используйте знак двойного равенства (например, if (x == 10)), который является оператором сравнения и проверяет, x равен 10 или нет. Последнее из двух выражений будет истинно, только если x равен 10, но предыдущее выражение всегда верно.

Это связано с тем, что C вычисляет выражение if (x=10) следующим образом: значение 10 присваивается x (помним, что простой знак равенства – это **оператор присваивания**), таким образом, x теперь равен 10. Затем условный **if** вычисляет 10, которое уже равно ИСТИНА, так как любое число, неравное 0, равно ИСТИНА. Поэтому if (x=10) будет всегда иметь логическое значение ИСТИНА, которое не является желательным результатом, когда используется оператор **if**. Вдобавок, переменной x будет присвоено значение 10, что также не является желаемым действием.

If также может быть частью разветвленной управляющей конструкции с использованием **if...else**

Функция If..else

Конструкция **if..else** предоставляет больший контроль над процессом выполнения кода, чем базовый оператор if, позволяя осуществлять несколько проверок, объединенных вместе. Например, аналоговый вход может быть проверен и выполнено одно действие, если на входе меньше 500, или другой действие, если на входе 500 или больше. Код при этом может выглядеть так:

```
if (pinFiveInput < 500) {
  // действие A
} else {
  // действие B
}
```

Другой способ создания переходов со взаимоисключающими проверками использует оператор switch case.

Else позволяет делать отличную от указанной в **if** проверку, чтобы можно было осуществлять сразу несколько взаимоисключающих проверок. Каждая проверка позволяет переходить к следующему за ней оператору не раньше, чем получит логический результат ИСТИНА. Когда проверка с результатом ИСТИНА найдена, запускается вложенная в нее блок операторов, и затем программа игнорирует все следующие строки в конструкции **if..else**. Если ни одна из проверок не получила результат ИСТИНА, по умолчанию выполняется блок операторов в **else**, если последний присутствует, и устанавливается действие по умолчанию.

Отметим, что конструкция **else if** может быть использована с или без заключительного **else** и наоборот. Допускается неограниченное число таких переходов **else if**.

```
if (pinFiveInput < 500){
  // выполнять действие A
}else if (pinFiveInput >= 1000){
  // выполнять действие B
}else{
  // выполнять действие  C
}
```

Другой способ создания переходов со взаимоисключающими проверками использует оператор switch case.

Оператор For

Конструкция **for** используется для повторения блока операторов, заключенных в фигурные скобки. Счетчик приращений обычно используется для приращения и завершения цикла. Оператор **for** подходит для любых повторяющихся действий и часто используется в сочетании с массивами коллекций данных/выводов.

Заголовок цикла **for** состоит из трех частей:

for (**initialization**; **condition**; **increment**) {операторы выполняющиеся в цикле}

Инициализация (Initialization) выполняется самой первой и один раз. Каждый раз в цикле проверяется условие (condition), если оно верно, выполняется блок операторов и приращение (increment), затем условие проверяется вновь. Когда логическое значение условия становится ложным, цикл завершается.

Пример

```
// Затемнение светодиода с использованием ШИМ-вывода
int PWMpin = 10; // Светодиод последовательно с резистором 470 ом на 10 выводов
void setup() {
 // настройка не нужна
}

void loop() {
   for (int i=0; i <= 255; i++){
      analogWrite(PWMpin, i);
      delay(10);
   }
}
```

Советы по применению

Цикл **for** в Си гораздо более гибкий, чем циклы **for** в других языках программирования, например, в Бейсике. Любой из трех или все три элемента заголовка могут быть опущены, хотя точки с запятой обязательны. Также операторы для инициализации, условия и приращения цикла могут быть любым допустимым в Си операторами с независимыми переменными, и использовать любой тип данных Си, включая данные с плавающей точкой (floats). Эти необычные для цикла **for** типы операторов позволяют обеспечить программное решение некоторых нестандартных проблем.

Например, использование умножения в операторе счетчика цикла позволяет создавать логарифмическую прогрессию:

```
for(int x = 2; x < 100; x = x * 1.5){
    println(x);
}
```

Генерируется: 2,3,4,6,9,13,19,28,42,63,94

Другой пример, плавное уменьшение или увеличение уровня сигнала на светодиод с помощью одного цикла **for:**

```
void loop(){
  int x = 1;
  for (int i = 0; i > -1; i = i + x){
      analogWrite(PWMpin, i);
      if (i == 255) x = -1;             // переключение
управления на максимуме
      delay(10);
   }
}
```

Смотреть также

while

Оператор Switch

Подобно конструкции **if**, **switch...case** управляет процессом выполнения программы, позволяя программисту задавать альтернативный код, который будет выполняться при разных условиях. В частности, оператор **switch** сравнивает значение переменной со значением, определенном в операторах case. Когда найден оператор **case**, значение которого равно значению переменной, выполняется программный код в этом операторе.

Ключевое слово **break** является командой выхода из оператора **case** и обычно используется в конце каждого **case**. Без оператора **break** оператор **switch**будет продолжать вычислять следующие выражения, пока не достигнет **break** или конец оператора **switch**.

Пример

```
switch (var) {
    case 1:
      //выполняется, когда var равно 1
      break;
    case 2:
      //выполняется когда  var равно 2
      break;
    default:
      // выполняется, если не выбрана ни одна альтернатива
      // default необязателен
  }
```

Синтаксис:

```
switch (var) {
 case label:
    // код для выполнения
    break;
 case label:
    // код для выполнения
    break;
 default:
    // код для выполнения
}
```

Параметры:

var: переменная, которая вычисляется для сравнения с вариантами в case
label: значение, с которым сравнивается значение переменной

Смотреть также:

if...else

Оператор While

While будет вычислять в цикле непрерывно и бесконечно до тех пор, пока выражение в круглых скобках, () не станет равно логическому ЛОЖНО. Что-то должно изменять значение проверяемой переменной, иначе выход из цикла **while** никогда не будет достигнут. Это изменение может происходить как в программном коде, например, при увеличении переменной, так и во внешних условиях, например, при тестировании датчика.

Синтаксис

```
while(выражение){
 // оператор(ы)
}
```

Параметры

выражение - (булевский) С-оператор, который возвращает значение истина или ложь

Пример

```
var = 0;

while(var < 200){
 // выполнить что-то, повторив 200 раз
 var++;
}
```

См. также

do ... while
break
continue
return

Оператор do ... while

Цикл **do** работает так же, как и цикл **while**, за исключением того, что условие проверяется в конце цикла, таким образом, цикл **do** будет *всегда* выполняться хотя бы раз.

```
do {
    // последовательность операторов
} while (проверка условия);
```

Пример

```
do {
  delay(50);           // подождать, пока датчики стабилизируются
  x = readSensors(); // проверить датчики
} while (x < 100);
```

См. также

while
break
continue
return

Break

Break используется для принудительного выхода из циклов **do, for** или **while**, не дожидаясь завершения цикла по условию. Он также используется для выхода из оператора **switch**

Пример

```
for (x = 0; x < 255; x ++)
{
    digitalWrite(PWMpin, x);
    sens = analogRead(sensorPin);
    if (sens > threshold){      // выходим из цикла если есть
сигнал с датчика
       x = 0;
       break;
    }
    delay(50);
}
```

См. также

[while](#)
[do ... while](#)
[continue](#)
[return](#)

continue

Оператор **continue** пропускает оставшиеся операторы в текущем шаге цикла. Вместо них выполняется проверка условного выражения цикла, которая происходит при каждой следующей итерации.

Пример

```
for (x = 0; x < 255; x ++){
    if (x > 40 && x < 120){       // если истина то прыгаем сразу
на следующую итерацию цикла
        continue;
    }
    digitalWrite(PWMpin, x);
    delay(50);

}
```

См. также

[while](#)
[do ... while](#)
[break](#)
[return](#)

return

Прекращает вычисления в функции и возвращает значение из прерванной функции в вызывающую, если это нужно.

Синтаксис

return;

return значение; // обе формы допустимы

Параметры

Значение: переменная или константа любого типа

Примеры:

Функция сравнивает значение на датчике входа с пороговым

```
int checkSensor(){
    if (analogRead(0) > 400) {
        return 1;
    else{
        return 0;
    }
}
```

С помощью ключевого слова return удобно тестировать блоки кода без «закомментирования» больших кусков с возможным ошибочным кодом.

```
void loop(){

  // здесь блестящая идея тестирования кода

  return;

  // оставшаяся часть неправильно функционирующего варианта
здесь
  // этот код никогда не будет выполняться

}
```

См. также

[while](#)
[do ... while](#)

break
continue

Оператор goto

Условное «перемещение» выполнения программы к определенной метке-указателю в самой программе, при этом пропускается весь код до самой метки, а исполняется - после нее.

Синтаксис:

```
label:
//
// какой-либо код
//
goto label; // переходим к метке label
```

Замечание по использованию

Использование **goto** не рекомендуется в С программировании, многие авторы книг не советуют его применять вообще, так как это не является необходимым (с их точки зрения). Причины их негодования заключаются в том, что программист при частом использовании в коде, команды **goto** - может запустить программу в бесконечный цикл, который потом трудно будет найти – отладка программы значительно усложниться. С другой стороны, если взгянуть на ассемблерный код, то там часто используется подобный переход по метке.

При разумном применении, команда может значительно упростить код программы и сохранить время программиста. Например, в случае необходимости выхода из глубоких циклов for, while, проверок if и прочих многократно вложенных контрукций.

Пример

```
for(byte r = 0; r < 255; r++){
    for(byte g = 255; g > -1; g--){
        for(byte b = 0; b < 255; b++){
            if (analogRead(0) > 250){ goto bailout;}
            // еще код
        }
    }
}
bailout:
```

Смотрите также

if

switch case
do.. while

; (точка с запятой)

Синтаксис **; (точка с запятой)** используется для обозначения конца оператора.

Пример

```
int a = 13;
```

Подсказка

Забытая в конце строки точка с запятой приводит к ошибке компиляции. Текст ошибки может быть либо видимым и ссылаться на пропущенную точку с запятой, либо нет. Если встречается непонятная или похожая на нелогичную ошибка компиляции, одним из первых действий должна быть проверка пропущенных точек с запятой, в коде, непосредственно предшествующем строке, в которой компилятор выдал предупреждение.

{} (Фигурные скобки)

Фигурные скобки **{}** (также называются просто «скобки») – важный элемент языка программирования С. Они используются в нескольких различных конструкциях, приведенных ниже, и это может иногда сбивать с толку начинающих.

Открывающая скобка "**{**" должна всегда сопровождаться закрывающей скобкой "**}**". Это условие, известное как парность (симметричность) фигурных скобок. Arduino IDE (интегрированная среда разработчика) включает подходящий инструмент для проверки парности скобок. Достаточно выделить скобку, или даже поставить курсор сразу же за скобкой, как будет подсвечена её логическая пара.

Сейчас эта возможность работает с ошибкой, так как IDE часто ищет (некорректно) скобку в тексте, который «закомментирован».

Начинающие программисты или программисты, перешедшие на Си с Бейсика, часто считают использование фигурных скобок сбивающим с толку или пугающим. В конце концов, одни и те же фигурные скобки заменяют оператор RETURN в подпрограммах (функциях), оператор ENDIF в условных циклах и оператор NEXT в циклах FOR.

Поскольку использование фигурных скобок столь многогранно, хорошей практикой программирования будет печатать закрывающую фигурную скобку сразу после того, как напечатана открывающая скобка, когда вставляется конструкция, для которой нужно использовать фигурные скобки. Затем возвращаем курсор в позицию между фигурными скобками и начинаем вводить операторы. Ваши скобки всегда будут парными и не лишат вас душевного равновесия.

Непарные скобки могут часто приводить к скрытым, непонятным ошибкам компиляции, которые сложно отследить в большой программе. Из-за их разного использования, скобки также невероятно важны в синтаксической правильности программы и перемещение скобки на одну или две строки часто приводят к значительному воздействию на логику программы.

Основные способы использования фигурных скобок

Функции

```
1          void НазваниеФункции(тип данных аргумента){
2             оператор(ы)
3          }
```

Циклы

```
while (логическое выражение) {
     оператор(ы)
 }
```

```
do{
   оператор(ы)
} while (логическое выражение);
```

```
for (инициализация; условие окончания цикла; приращения ци
   оператор(ы)
}
```

Условные операторы

```
if (логическое выражение){
   оператор(ы)
}else if (логическое выражение){
   оператор(ы)
}else{
   оператор(ы)
}
```

Комментарии

Комментарии – это строки в программе, которые используются для информирования вас самих или других о том, как работает программа. Они игнорируются компилятором и не экспортируются в процессор, таким образом, они не занимают место в памяти микроконтроллера Atmega.

Комментарии предназначены только для того, чтобы помочь вам понять (или вспомнить), как работает ваша программа или объяснить это другим. Есть два способа пометить строку как комментарий:

Пример

```
x = 5; // Это комментарий в одной строке. Все после двойного
слэша – комментарий
      // до конца строки

/* это многострочный комментарий – используйте его для
закоментирования целых кусков кода

if (gwb == 0){   // комментарий в строке допустим внутри
многострочного комментария
                 // но не другой многострочный комментарий

}
// не забывайте «закрывать» комментарии – они должны быть
парными!
*/
```

Подсказка

Во время экспериментов с кодом, «закомментирование» частей программы – подходящий способ удаления строк, в которых могут быть ошибки. Так строки в коде остаются, но превращаются в комментарии, и компилятор просто игнорирует их. Это может быть особенно полезно при локализации проблемы, или когда не получается скомпилировать программу, а сообщение об ошибке при компиляции скрыто или бесполезно.

Комментарии

Комментарии – это строки в программе, которые используются для информирования вас самих или других о том, как работает программа. Они игнорируются компилятором и не экспортируются в процессор, таким образом, они не занимают место в памяти микроконтроллера Atmega.

Комментарии предназначены только для того, чтобы помочь вам понять (или вспомнить), как работает ваша программа или объяснить это другим. Есть два способа пометить строку как комментарий:

Пример

```
x = 5; // Это комментарий в одной строке. Все после двойного
слэша – комментарий
      // до конца строки

/* это многострочный комментарий – используйте его для
закоментирования целых кусков кода

if (gwb == 0){   // комментарий в строке допустим внутри
многострочного комментария
                 // но не другой многострочный комментарий

}
// не забывайте «закрывать» комментарии – они должны быть
парными!
*/
```

Подсказка

Во время экспериментов с кодом, «закомментирование» частей программы – подходящий способ удаления строк, в которых могут быть ошибки. Так строки в коде остаются, но превращаются в комментарии, и компилятор просто игнорирует их. Это может быть особенно полезно при локализации проблемы, или когда не получается скомпилировать программу, а сообщение об ошибке при компиляции скрыто или бесполезно.

Директива #define

#define это удобная директива, который позволяет дать имя константе перед тем как программа будет скомпилирована. Определенные этой директивой константы не занимают программной памяти, поскольку компилятор заменяет все обращения к ним их значениями на этапе компиляции, соответственно они служат исключительно для удобства программиста и улучшения читаемости текста программы.

Стоит упомянуть о некотором нежелательном эффекте, который может иметь место при использовании директивы #define. Например, если имя константы, заданное с помощью директивы #define включить в имя другой константы или переменной, то оно будет заменено на свое значение.

В общем случае рекомендуется использовать выражение const для определения констант вместо #define

Синтаксис для Arduino такой же как и для C:

Синтаксис:

```
#define constantName value
```
Внимание! Символ # перед словом define обязателен.

Пример

```
#define ledPin 3
// компилятор заменит любое упоминание ledPin на занчение 3 во время компиляции
```

Замечание по использованию

Обратите внимание, что точка с запятой не ставится, иначе компилятор выдаст критическую ошибку.
```
#define ledPin 3;    // это ошибка, ; здесь не нужна
```
Точно так же знак равно после имени константы тоже вызовет критическую ошибку компилятора.
```
#define ledPin  = 3  // это тоже ошибка, знак = не нужен
```

Смотрите также

const
Константы

Директива #include

#include используется для включчения сторонних библиотек в ваш скетч. Это дает доступ к большому числу стандартных библиотек C (библиотекой называют группы предварительно написанных функций), и библиотек написанных специально для Arduino.

Здесь можно ознакомиться с C библиотеками для AVR (AVR это тип микроконтроллеров фирмы Atmel, на которых построено большинство плат Arduino).

Обратите внимание, что #include, так же как и #define, не требует точки запятой в конце, если же ее добавить компилятор выдаст кртическую ошшибку.

Пример

Этот пример подключает библиотеку, которая используется, чтобы помещать данные в область программ вместо RAM. Это сохраняет место RAM для нужд динамической памяти и делает большие справочные таблицы более практичными.

```
#include <avr/pgmspace.h>
prog_uint16_t myConstants[] PROGMEM = {0, 21140, 702  , 9128,
0, 25764, 8456,
0,0,0,0,0,0,0,0,29810,8968,29762,29762,4500};
```

= оператор присваивания

Присваивает переменной слева от оператора значение переменной или выражения, находящееся справа.

Пример

```
int sensVal;                  // объявление переменной типа
integer
senVal = analogRead(0);        // присваивание переменной
sensVal, значения, считанное с аналогового входа 0
```

Важно

Переменная слева от оператора присваивания (=) должна быть способна сохранить присваиваемое значение. Если оно выходит за диапазон допустимых значений, то сохраненное значение будет не верно.

Необходимо различать оператор присваивания (=) и оператор сравнения (== двойной знак равенства), который осуществляет проверку на равенство.

Смотрите также

Операторы сравнения, if
int
long
char

Сложение, вычитание, умножение и деление

Операторы **+**, **-**, ***** и **/** соответственно, возвращают результат выполнения арифметических действий над двумя операндами. Возвращаемый результат будет зависеть от типа данных операндов, например, 9 / 4 возвратит 2, т.к. операнды 9 и 4 имеют тип **int**. Также следует следить за тем, чтобы результат не вышел за диапазон допустимых значений для используемого типа данных. Так, например, сложение 1 с переменной типа **int** и значением 32 767 возвратит -32 768. Если операнды имеют разные типы, то тип с более "широким" диапазоном будет использован для вычислений.

Если один из операндов имеет тип float или double, то арифметика "с плавающей запятой" будет использована для вычислений.

Пример

```
y = y + 3;
x = x - 7;
i = j * 6;
r = r / 5;
```

Синтаксис

```
result = value1 + value2;
result = value1 - value2;
result = value1 * value2;
result = value1 / value2;
```

Параметры

value1: любая переменная или константа
value2: любая переменная или константа

Советы по использованию

Помните, что целочисленные константы воспринимаются компилятором как тип **int**, следите за вхождением результата в диапазон допустимых значений
Вычисления с "плавающей запятой" выполняются дольше чем целочисленные.

% оператор

Возвращает остаток от деления одного целого (**int**) операнда на другой.

Синтаксис

```
result = dividend % divisor
```

Параметры

dividend: делимое
divisor: делитель

Возвращаемое значение

Остаток от деления.

Пример

```
x = 7 % 5;    // x имеет значение 2
x = 9 % 5;    // x имеет значение 4
x = 5 % 5;    // x имеет значение 0
x = 4 % 5;    // x имеет значение 4
```

Советы по использованию %

Нельзя применить к типу float.

Оператор If

if (условие) и ==, !=, <, > (операторы сравнения)

if, используется в сочетании с операторами сравнения, проверяет, достигнута ли истинность условия, например, превышает ли входное значение заданное число. Формат оператора **if** следующий:

```
if (someVariable > 50)
{
// выполнять действия
}
```

Программа проверяет, значение *someVariable* больше чем 50 или нет. Если да, то выполняются определенные действия. Говоря иначе, если выражение в круглых скобках истинно, выполняются операторы внутри фигурных скобок. Если нет, программа пропускает этот код.

Скобки после оператора **if** могут быть опущены. Если так сделано, только следующая строка (обозначенная точкой с запятой) становится оператором, выполняемым в операторе **if**.

```
if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120)
digitalWrite(LEDpin, HIGH);

if (x > 120){ digitalWrite(LEDpin, HIGH); }

if (x > 120){
digitalWrite(LEDpin1, HIGH);
digitalWrite(LEDpin2, HIGH);
}                                    // все правильно
```

Выражения, которые вычисляются внутри круглых скобок, могут состоять из одного или нескольких операторов.

Операторы сравнения

x == y (x равно y)

x != y (x не равно y)

x < y (x меньше чем y)

x > y (x больше чем y)

x <= y (x меньше чем или равно y)

x >= y (x больше чем или равно y)

Внимание!

Следите, чтобы случайно не использовать знак простого равенства (например, if (x = 10)). Знак простого равенства – это оператор присваивания, и устанавливает значение x равное 10 (заносит значение 10 в переменную x). Вместо этого используйте знак двойного равенства (например, if (x == 10)), который является оператором сравнения и проверяет, x равен 10 или нет. Последнее из двух выражений будет истинно, только если x равен 10, но предыдущее выражение всегда верно.

Это связано с тем, что C вычисляет выражение if (x=10) следующим образом: значение 10 присваивается x (помним, что простой знак равенства – это **оператор присваивания**), таким образом, x теперь равен 10. Затем условный **if** вычисляет 10, которое уже равно ИСТИНА, так как любое число, неравное 0, равно ИСТИНА. Поэтому if (x=10) будет всегда иметь логическое значение ИСТИНА, которое не является желательным результатом, когда используется оператор **if**. Вдобавок, переменной x будет присвоено значение 10, что также не является желаемым действием.

If также может быть частью разветвленной управляющей конструкции с использованием **if...else**

Логические операторы

Логические операторы чаще всего используются в проверке условия оператора if. Базовые сведения о логических операциях, смотрите в Википедии.

&& (логическое И)

Истина, если оба операнда истина (true).

```
if (digitalRead(2) == HIGH  && digitalRead(3) == HIGH) { // считывает состояние двух портов
  // ...
}
```

Истина если оба порта вход/выхода HIGH

|| (логическое ИЛИ)

Истина, если хотя бы один операнд истина, например:

```
if (x > 0 || y > 0) {
  // ...
}
```

будет верно (истина) если x или y больше 0.

! (логическое отрицание)

True, если операнд false, и наоборот, например:

```
if (!x) {
  // ...
}
```

условие верно, если x - false (x равно 0).

Важно различать логический оператор "И" - **&&** и битовый оператор "И" - **&**. Тоже самое относится к логическому оператору "ИЛИ" - **||** и битовому оператору "ИЛИ" - **|**.

Пример

```
if (a >= 10 && a <= 20){}   // условие верно, если a больше 10, но меньше 20
```

Указатели доступа

& (ссылка) and * (указатель)

Указатели являются одним из наиболее сложных предметов для новичков в изучении C. Можно писать подавляющее большинство скетчей Arduino, никогда не использую указателей. Однако для работы с определенными структурами данных, использование указателей может упростить код.

Побитовое И (&), Побитовое ИЛИ (|), Сложение по модулю два (^)

Битовые операторы выполняют свои расчеты на уровне битов переменных. Они помогают решать широкий круг общих проблем программирования. Хорошую статью по битовым операциям можно найти на Википедии

Побитовое И (&)

В C++ оператор побитового И указывается одиночным амперсандом, он ставится между двумя целыми выражениями. Побитовое И действует на позиции каждого бита, окружающих выражения независимо от того какой из операндов стоит первым, а какой вторым. В соответствии с правилом: если оба входных бита равны 1, результирующий выходной сигнал равен 1, в противном случае выход равен 0. Иллюстрация:

```
0   0   1   1    операнд1
0   1   0   1    операнд2
----------
0   0   0   1    результат (операнд1 & операнд2)
```

В среде разработки Arduino тип int это 16-битное значение, таким образом оператор побитовое И между двумя int выражениями делает 16 одновременных И операций. Иллюстрация:

```
int a =  92;    // в двоичном виде: 0000000001011100
int b = 101;    // в двоичном виде: 0000000001100101
int c = a & b;  // результат:    0000000001000100, или 68 в десятичном представлении
```

Каждый бит значений a и b проходит операцию побитового И, в результате все 16 битов попадают в переменную c, полученное значение будет 01000100, что равносильно 65 в двоичном представлении.
Наиболее часто операция побитового И используется для выбора конкретного бит (или битов) от целого значения, часто называемая маска. См. пример ниже.

Побитовое ИЛИ (|)

Побитовое ИЛИ в C++ обозначается вертикальной чертой, |. Как и оператор &, оператор | работает независимо с каждым битом окружающиъ его чисел. Результат операции побитового ИЛИ двух бит будет 1, если хотя бы один из этих битов 1, иначе резульат будет 0. Другими словами:

```
0   0   1   1    операнд1
0   1   0   1    операнд2
----------
```

```
    0  1  1  1    (операнд1 | операнд2) - результат
```

Пример использования операции побитового ИЛИ в фрагменте кода на C++:

```
int a =  92;    // в бинарном виде: 0000000001011100
int b = 101;    // в бинарном виде: 0000000001100101
int c = a | b;  // результат:    0000000001111101, или 125 в
десятичном виде.
```

Пример программы

Основная область примернения операторов побитовое И/ИЛИ это операции чтение/записи в порт. В микроконтроллерах порт это 8-битовое число, через которое можно получить информацию о состоянии контактов. Записи в порт контролирует все контакты сразу.

PORTD является встроенной константой, которая относится к выходному состоянию цифровых выводов 0,1,2,3,4,5,6,7. Если в каком-то бите установлена 1, значит на выводе состояние HIGH. (Вывод при этом должен быть установлен как выход командой pinMode().) Таким образом, если записать `PORTD = B00110001;` мы установим выводы 2,3 и 7 в состояние логической 1.

Наш алгоритм будет выглядеть вот так:
Получить значение PORTD и очистить биты относящиеся к выводам, значение которые мы хотим изменить (с помощью побитового И).
Скомбинировать значение PORTD с нашим новым значением (через операцию побитового ИЛИ).

```
int i;      // переменная счетчика
int j;

void setup(){
DDRD = DDRD | B11111100; // установить биты направлений для
выводов с 2 по 7, оставить нетронутыми для выводов 0 и 1 (xx |
00 == xx)
// это тоже самое что и оператор pinMode(pin, OUTPUT) для
выводов с 2 по 7
Serial.begin(9600);
}

void loop(){
for (i=0; i<64; i++){

PORTD = PORTD & B00000011;  // очистить биты с 2 по 7, оставить
биты (а занчить и выводы) 0 и 1 неизмененными (xx & 11 == xx)
j = (i << 2);               // сдвинуть переменную на два бита
влево к пинам 2 -7, чтобы не затронуть пины 0 и 1
```

```
PORTD = PORTD | j;          // скомбинирвать значение порта с
новым значением
Serial.println(PORTD, BIN); // для отладки выведем значение
порта в терминал
delay(100);
   }
}
```

побитовое XOR или исключающее ИЛИ (^)

Еще один побитовый оператор, немного похожий на обычное бинарное ИЛИ, но небольшим отлиием - он вернет 0, если оба бита будут равны 1. Обозначается символом ^.

```
    0  0  1  1    операнд1
    0  1  0  1    операнд2
    ----------
    0  1  1  0    (операнд1 ^ операнд2) - результат
```

Другими словами, после операции илсключающего ИЛИ будет 1 в том случае, если входные биты различны или оба равны 0.

Пример кода:

```
int x = 12;      // в бинарном представлении: 1100
int y = 10;      // в бинарном представлении: 1010
int z = x ^ y;  // в бинарном представлении: 0110, или десятчиная
6
```

Оператор ^ часто используется для переключения (т.е. чтобы изменить 0 на 1 или 1 на 0) каких-либо битов в цифровом представлении. Пример программы для переключения цифрового вывода 5.

```
void setup(){
DDRD = DDRD | B00100000; // установили цифровой вывод 5 как
выход
Serial.begin(9600);
}
void loop(){
PORTD = PORTD ^ B00100000;  // переключили бит 5 (цифровой вывод
5), остальные оставили нетронутыми
delay(100);
}
```

Побитовое НЕ (~)

В C++ оператор побитового отрицания обозначается тильдой ~. Этот оператор, в отличие от & и |, употребляется применительно к одному операнду, который указывается после ~. Побитовое НЕ меняет каждый бит операнда на противоположный: 0 становится 1, а 1 становится 0. Например:

```
0 1 operand1

----------
1 0 ~ operand1

int a = 103; // в двоичной системе: 0000000001100111
int b = ~a; // в двоичной системе: 1111111110011000 = -104
```

Вас может удивить, что результатом данной операции является отрицательное число -104. Это объясняется тем, что старший бит в переменной типа int является так называемым знаковым битом. Если старший бит 1, то число считается отрицательным. Такое представление положительных и отрицательных чисел принято называть дополнительным кодом. Более подробную информацию см. в статье Википедии "Дополнительный код".

Примечательно, что для любого целочисленного x, результатом операции ~x будет являться число (-x-1).

Таким образом, в выражениях со знаковыми целыми числами знаковый бит иногда может приводить к неожиданным, на первый взгляд, результатам.

Побитовый свдиг влево (<<), побитовый сдвиг вправо (>>)

Описание

В C++ есть два оператора побитового сдвига: оператор сдвига влево << и оператор сдвига вправо >>. Эти операторы заставляют биты левого операнда сдвинуться влево или вправо на то количество позиций, которое указано во втором операнде.

Синтаксис

переменная << количество_бит
переменная >> количество_бит

Параметры

переменная - (byte, int, long)
количество_бит - целое число <= 32

Пример:

```
int a = 5;         // в двоичной системе: 0000000000000101
int b = a << 3;    // в двоичной системе: 0000000000101000, или 40 в десятичной
int c = b >> 3;    // в двоичной системе: 0000000000000101, или снова 5, как было изначально
```

Следует иметь ввиду, что при сдвиге значения x на y бит (x<<y), самые левые y бит в исходном числе x теряются, т.к. они буквально выталкиваются за его пределы.

```
int a = 5;         // в двоичной системе: 0000000000000101
int b = a << 14;   // в двоичной системе: 0100000000000000 - первая 1 в 101 исчезла
```

Если вы уверены, что ни один из битов в сдвигаемом числе не пропадет, то для простоты можно считать, что оператор сдвига << умножает левый операнд на 2 в степени, показателем которой является правый операнд. Например, для получения степеней 2 могут быть использованы следующие выражения:

```
1 <<  0  ==    1
1 <<  1  ==    2
```

```
1 <<  2  ==    4
1 <<  3  ==    8
...
1 <<  8  ==  256
1 <<  9  ==  512
1 << 10  == 1024
...
```

Если вы сдвигаете x вправо на y бит (x>>y) и при этом старшим битом x является 1, то результат такой операции будеть зависеть от типа переменной x. Как уже отмечалось ранее, в переменных типа int старший бит является знаковым битом, определяющим является ли число положительным или отрицательным. Если переменная x имеет тип int, то при сдвиге x вправо знаковый бит копируется в младшие биты (по историческим причинам):

```
int x = -16;     // в двоичной системе: 1111111111110000
int y = x >> 3;  // в двоичной системе: 1111111111111110
```

Такое поведение называется расширением знака и, как правило, нежелательно: вместо единиц пользователь чаще ожидает увидеть нули в левой части x на месте свдинутых бит. В то же время для беззнаковых целых чисел (переменные типа unsigned int) действуют другие правила сдвига вправо. Поэтому для предотвращения копирования единиц в старших разрядах сдвигаемой переменной x, можно прибегнуть к преобразованию типов:

```
int x = -16;                    // в двоичной системе:
1111111111110000
int y = (unsigned int)x >> 3;  // в двоичной системе:
0001111111111110
```

Таким образом, если предотвращать эффект расширения знака, оператор сдвига вправо >> можно использовать для деления числа на степени 2. Например:

```
int x = 1000;
int y = x >> 3;   // целочисленное деление 1000 на 8, в
результате которого y = 125.
```

++ (инкремент) / -- (декремент)

Описание

Инкрементирует или декрементирует значение переменной

Синтаксис

```
x++;  // увеличивает значение x на 1 и возвращает старое
значение x
++x;  // увеличивает значение x на 1 и возвращает новое значение
x

x-- ;   // уменьшает значение x на 1 и возвращает старое
значение x
--x ;   // уменьшает x на 1 и возвращает новое значение x
```

Параметры

x: целое int или long (беззнаковые типы допускаются)

Возвращаемые значения

Исходное или новое инкрементированное / декрементированное значение переменной.

Примеры

```
x = 2;
y = ++x;      // x имеет значение 3, y содержит 3
y = x--;      // x снова имеет значение 2, y по прежнему
содержит 3
```

Смотрите также

+=
-=

+=, -=, *=, /=

Описание

Осуществляет математическую операцию между двумя переменными. Оператор += (и др.) является простой сокращенной формой математических выражений, перечисленных ниже:

Синтаксис

```
x += y;   // эквивалентно выражению x = x + y;
x -= y;   // эквивалентно выражению x = x - y;
x *= y;   // эквивалентно выражению x = x * y;
x /= y;   // эквивалентно выражению x = x / y;
```

Параметры

x: переменная любого типа

y: переменная любого типа или константа

Примеры

```
x = 2;
x += 4;       // x содержит 6
x -= 3;       // x содержит 3
x *= 10;      // x содержит 30
x /= 2;       // x содержит 15
```

Составное побитовое И (&=)

Описание

Оператор составного побитового И (&=) часто употребляется между переменной и константой чтобы перевести отдельные биты переменной в низкий уровень (0). В программировании эту операцию часто называют "очисткой" или "сбросом" бит.

Синтаксис

```
x &= y;   // эквивалентно x = x & y;
```

Параметры

x: переменная типа char, int или long

y: целочисленная константа либо переменная типа char, int или long

Пример:

Для начала рассмотрим действие оператора побитового И (&)

```
0  0  1  1    операнд1
0  1  0  1    операнд2
----------
0  0  0  1    (операнд1 & операнд2) - возвращаемый результат
```

При выполнении операции побитового И, биты операнда1, взаимодействующие с 0 операнда2, очищаются. Таким образом, если myByte - переменная типа byte,

```
myByte & B00000000 = 0;
```

При выполнении операции побитового И, биты операнда1, взаимодействующие с 1, остаются неизменными, поэтому:

```
myByte & B11111111 = myByte;
```

Примечание: поскольку в побитовых операторах мы имеем дело с битами - удобнее использовать двоичное представление констант. В других системах счисления они являются точно такими же числами, но не так просты для понимания. Число B0000000 показано для ясности, хотя ноль в любой системе счисления является нулем (хммм, здесь есть что-то философское, не правда ли?)

Следовательно, чтобы очистить (привести к 0) 0-й и 1-й биты переменной, не затронув при этом остальные, необходимо использовать оператор

составного побитового И (&=) с константой B11111100:

```
  1  0  1  0  1  0  1  0    переменная
  1  1  1  1  1  1  0  0    маска
  ---------------------
  1  0  1  0  1  0  0  0

неизменившиеся биты
                     очищенные биты
```

Если биты переменной обозначить как x, то та же операция будет выглядеть так:

```
   x  x  x  x  x  x  x  x    переменная
   1  1  1  1  1  1  0  0    маска
   ---------------------
   x  x  x  x  x  x  0  0

неизменившиеся биты
                      очищенные биты
```

Поэтому, если:

```
myByte =  10101010;

myByte &= B1111100 == B10101000;
```

Смотрите также

|= (составное побитовое ИЛИ)
& (побитовое И)
| (побитовое ИЛИ)

Составное побитовое ИЛИ (|=)

Описание

Оператор составного побитового ИЛИ (|=) часто употребляется между переменной и константой чтобы установить (перевести в 1) отдельные биты переменной.

Синтаксис

```
x |= y;   // эквивалентно x = x | y;
```

Параметры

x: переменная типа char, int или long

y: целочисленная константа либо переменная типа char, int или long

Пример

Для начала рассмотрим действие оператора побитового ИЛИ (|):

```
0  0  1  1    операнд1
0  1  0  1    операнд2
----------
0  1  1  1    (операнд1 | операнд2) - возвращаемый результат
```

При выполнении операции побитового ИЛИ, биты операнда1, взаимодействующие с 0 операнда2, не изменяются. Поэтому, если myByte - переменная типа byte,

```
myByte | B00000000 = myByte;
```

При выполнении операции побитового ИЛИ, биты операнда1, взаимодействующие с 1, устанавливаются в 1:

```
myByte | B11111111 = B11111111;
```

Следовательно, чтобы установить 0-й и 1-й биты переменной, не затронув при этом остальные, необходимо использовать оператор составного побитового ИЛИ (|=) с константой B00000011

```
  1  0  1  0  1  0  1  0    переменная
  0  0  0  0  0  0  1  1    маска
  ----------------------
  1  0  1  0  1  0  1  1

неизменившиеся биты
                     установленные биты
```

Если биты переменной обозначить как x, то та же операция будет выглядеть так:

```
   x  x  x  x  x  x  x  x    переменная
   0  0  0  0  0  0  1  1    маска
   ----------------------
   x  x  x  x  x  x  1  1

неизменившиеся биты
                     установленные биты
```

Поэтому, если:

```
myByte =  B10101010;

myByte |= B00000011 == B10101011;
```

Смотрите также

[&= (составное побитовое И)]
[& (побитовое И)]
[| (побитовое ИЛИ)]

Константы

В языке Ардуино константы - это предопределенные переменные. Они используются для улучшения читабельности программного кода. Все константы можно условно разделить на несколько групп.

Константы, характеризующие логические уровни, true или false (Булевы константы)

В языке Ардуино существует две константы, использующиеся для обозначения истинности или ложности: **true** и **false**.

false

Наиболее проста в определении константа false. false означает 0 (ноль).

true

Часто считают, что константа true означает 1, что является верным, однако true имеет более широкое значение. Любое целое число, не равное 0, логически является истиной (true). Поэтому числа -1, 2 и -200 в Булевой алгебре будут также считаться истиной (true).

Обратите внимание, что константы *true* и *false* пишутся в нижнем регистре, в отличие от констант HIGH, LOW, INPUT и OUTPUT.

Константы, характеризующие уровень напряжения на выводах, HIGH и LOW

При работе с цифровыми выводами существует всего два значения, которые они могут выводить или считывать: **HIGH** и **LOW**.

HIGH

Понятие HIGH (применительно к выводу) может несколько отличаться в зависимости от того, как настроен вывод - как вход (INPUT) или как выход (OUTPUT). Если функцией pinMode вывод сконфигурирован как вход (INPUT), то при считывании с него данных (функция digitalRead) микроконтроллер ответит HIGH в том случае, когда на выводе присутствует напряжение 3В или больше.

Также возможна ситуация, когда функцией pinMode вывод сконфигурирован как вход (INPUT), после чего функцией digitalWrite на него подается

высокий уровень HIGH. В этом случае к выводу будут подключены внутренние подтягивающие резисторы номиналом 20 кОм, что приведет к возникновению на нем высокого уровня HIGH. При считывании значение HIGH будет удерживаться до тех пор, пока внешними цепями на выводе не будет сформирован низкий уровень LOW. Именно так работает режим INPUT_PULLUP.

Если функцией pinMode вывод сконфигурирован как выход (OUTPUT) и функцией digitalWrite на него подан высокий уровень HIGH, то на выводе установится напряжение 5В. В этом режиме он может быть источником тока и, например, засвечивать светодиод, последовательно подключенный через резистор к земле либо к другому выходу с уровнем LOW.

LOW

Понятие LOW также имеет разные значения в зависимости от того, как настроен вывод - как вход (INPUT) или выход (OUTPUT). Если функцией pinMode вывод сконфигурирован как вход (INPUT), то при считывании с него данных функцией digitalRead микроконтроллер ответит LOW в том случае, когда напряжение на выводе не превышает 2В.

Если функцией pinMode вывод сконфигурирован как выход (OUTPUT) и функцией digitalWrite на него подан низкий уровень LOW, то на выводе установится напряжение 0В. В этом режиме он может принимать втекающий ток, например от светодиода, подключенного через резистор к +5В либо к другому выходу с уровнем HIGH.

Константы, характеризующие цифровые выводы, INPUT, INPUT_PULLUP и OUTPUT

Выводы, сконфигурированные как INPUT

Выводы Ардуино (ATmega), сконфигурированные функцией pinMode() как входы (INPUT), находятся в высокоимпедансном состоянии. Это эквивалентно подключению к выводу последовательного резистора в 100 МОм, поэтому к цепям, подключенным к таким выводам, не предъявляется практически никаких требований. Такой режим удобен для считывания сигналов с датчиков, но не приемлем для питания светодиодов.

Следует отметить, что входы INPUT иногда соединяют с землей через подтягивающий резистор (резистор на землю), как описано в примере использования последовательном связи.

Выводы, сконфигурированные как INPUT_PULLUP

Микроконтроллер ATmega в Ардуино имеет внутренние подтягивающие резисторы (резисторы, подключенные к питанию внутри микросхемы), которыми можно управлять. Если вы предпочитаете использовать их вместо внешних резисторов, подключенных к земле, - используйте параметр INPUT_PULLUP в функции pinMode(). Это позволит инвертировать поведение подключенного к выводу внешнего датчика: HIGH будет означать его отключение, а LOW - включение. См. пример использования INPUT_PULLUP при последовательной связи.

Выводы, сконфигурированные как OUTPUT

Выводы, сконфигурированные функцией pinMode() как выходы (OUTPUT), находятся в низкоимпедансном состоянии. Это означает, что они могут обеспечить внешние цепи относительно большим током. Микроконтроллер ATmega может отдавать (положительный ток) или принимать (отрицательный) ток до 40 мА (миллиампер) от внешних устройств/цепей. Такой режим удобен для питания светодиодов, но бесполезен при считывании сигналов с датчиков. Выводы, сконфигурированные как выход, также могут быть выведены из строя при коротком замыкании на землю либо на цепь питания 5В. Кроме того, выходного тока микроконтроллера ATmega недостаточно для питания большинства реле и двигателей, что требует дополнительных интерфейсных цепей.

Смотрите также

pinMode()
целочисленные константы
переменные boolean

Целочисленные константы

Целочисленные константы в коде - это набираемые числа, например, 123. По умолчанию эти числа интерпретируются как целые типа int, однако вы можете изменить это с помощью модификаторов U и L (см. ниже).

Целочисленные константы интерпретируются как числа в десятичной системе счисления, поэтому для задания числа в другой системе необходимо использовать специальные префиксы.

Основание системы	Пример	Префикс	Комментарий
10 (десятичная)	123	нет	
2 (двоичная)	B1111011	"B"	работает только с 8-битными значениями (0 - 255)
			корректные символы 0-1
8 (восьмеричная)	0173	"0"	корректные символы 0-7
16 (шестнадцатиричная)	0x7B	"0x"	корректные символы 0-9, A-F, a-f

Основание **десятичной** системы счисления - 10. Математические операции с такими числами всем знакомы. Константы без каких-либо префиксов считаются десятичными.

Пример:

```
101     // то же, что и 101 в десятичной системе   ((1 * 10^2) + (0 * 10^1) + 1)
```

Основание **двоичной** системы счисления - 2. Для записи чисел в этой системе используются только 0 и 1.

Пример:

```
B101    // то же, что и 5 в десятичной системе   ((1 * 2^2) + (0 * 2^1) + 1)
```

Префикс двоичной системы может использоваться только с числами размерностью 1 байт (8 бит) в диапазоне от 0 (B0) до 255 (B11111111). Для записи числа типа int (16 бит) в двоичном формате можно использовать двойную операцию:

```
myInt = (B11001100 * 256) + B10101010;    // B11001100 - старший байт
```

Основание **восьмеричной** системы счисления - 8. Для записи чисел в этой системе используются только цифры в диапазоне от 0 до 7. Восьмеричные числа характеризуются префиксом "0".

Пример:

```
0101    // то же, что и 65 в десятичной системе   ((1 * 8^2) +
```

```
(0 * 8^1) + 1)
```

Внимание. Можно допустить труднонаходимую ошибку, если нечаянно дописать 0 перед константой, поскольку в этом случае последняя будет интерпретироваться компилятором как восьмеричное число.

Основание **шестнадцатеричной** системы счисления - 16. Для записи чисел в этой системе используются цифры от 0 до 9, а также буквы от A до F; при этом A имеет значение 10, B - 11, и т.д. до F, которое эквивалентно 15. Шестнадцатиричные значения характеризуются префиксом "0x". Обратите внимание, что A-F могут вводится как в верхнем, так и нижнем регистрах (a-f).

Пример:

```
0x101   // то же, что и 257 в десятичной системе   ((1 * 16^2) + (0 * 16^1) + 1)
```

U & L модификаторы

По умолчанию, целочисленные константы интерпретируются как целые числа типа int с соответствующими предельными значениями. Чтобы задать целочисленной константе другой тип, запишите после нее:

'u' или 'U", чтобы привести константу к беззнаковому типу данных. Например: 33u
'l' или 'L', чтобы привести константу к типу данных long. Например: 100000L
'ul' или 'UL', чтобы привести константу к типу unsigned long. Например: 32767ul

Смотрите также

константы
#define
byte
int
unsigned int
long
unsigned long

Константы с плавающей точкой

Подобно целочисленным константам, константы с плавающей точкой используются для того, чтобы сделать программный код более читабельным. При компиляции кода дробные константы заменяются их числовыми значениями.

Например:

```
n = .005;
```

Константы с плавающей точкой также могут быть выражены в экспоненциальной форме. Оба символа 'E' и 'e' интерпретируются как показатели экспоненты.

Константа с плавающей точкой	принимает значение:	также эквивалентно:
10.0	10	
2.34E5	2.34 * 10^5	234000
67e-12	67.0 * 10^-12	.000000000067

void

Ключевое слово void используется только при объявлении функций. Оно указывает на то, что объявляемая функция не возвращает никакого значения той функции, из которой она была вызвана.

Пример:

```
// операции выполняются в функциях "setup" и "loop"
// но общей программе никакой информации не возвращается

void setup()
{
  // ...
}

void loop()
{
  // ...
}
```

boolean

Переменные типа **boolean** могут принимать одно из двух значений: true или false. (Каждая переменная типа boolean занимает в памяти один байт.)

Пример:

```
int LEDpin = 5;        // светодиод подключен к выводу 5
int switchPin = 13;    // ключ подключен к выводу 13, другой вывод
подключен к земле

boolean running = false;

void setup()
{
  pinMode(LEDpin, OUTPUT);
  pinMode(switchPin, INPUT);
  digitalWrite(switchPin, HIGH);       // включить подтягивающий
резистор
}

void loop()
{
  if (digitalRead(switchPin) == LOW)
  {  // ключ нажат - вывод подтянут к высокому уровню сигнала
    delay(100);                          // задержка для
устранения дребезга контактов ключа
    running = !running;                  // инвертирование
переменной running
    digitalWrite(LEDpin, running)        // индикация светодиодом
  }
}
```

Смотрите также

константы
логические операторы
Объявление переменных

char

Описание

Тип данных, который занимает в памяти 1 байт и хранит символьное значение. Символы пишутся в одинарных кавычках, например: 'A' (совокупность символов - строки - пишутся в двойых кавычках: "ABC").

Однако в памяти символы хранятся как числа. Соответствующую кодировку вы можете найти в таблице ASCII символов. Поэтому, над символами можно совершать арифметические операции, при вычислении которых будет использоваться ASCII-код символа (например, 'A' + 1 будет равно 66, поскольку ASCII-код большой буквы A - 65). Более подробно о том, как символы переводятся в числа, смотрите раздел Serial.println.

Тип данных char - это знаковый тип, т.е. переменные данного типа могут хранить числовые значения в диапазоне от -128 до 127. В качестве беззнакового однобайтового (8 бит) типа данных используйте тип данных byte.

Пример:

```
char myChar = 'A';
char myChar = 65;       // оба значения эквивалентны
```

Смотрите также

byte
int
array
Serial.println

unsigned char

Описание

Беззнаковый тип данных, занимающий в памяти 1 байт. То же самое, что и тип данных byte.

Переменные типа unsigned char могут хранить значения в диапазоне от 0 до 255.

В соответствии со стилем программирования Ардуино, тип *byte* является более предпочтительным.

Пример:

```
unsigned char myChar = 240;
```

Смотрите также

byte
int
array
Serial.println

byte

Описание

Переменная типа byte хранит 8-битное беззнаковое число, от 0 до 255.

Пример:

```
byte b = B10010;  // "B" - префикс двоичной системы счисления
(B10010 = 18 в десятичной системе)
```

Смотрите также

word
byte()
Объявление переменных

int

Описание

Целочисленный тип *int* - это основной тип данных для хранения чисел.

В Arduino Uno (и других платах на базе микроконтроллеров ATmega) переменные типа *int* хранят 16-битные (2-байтовые) значения. Такая размерность дает диапазон от -32768 до 32767 (минимальное значение -2^15 и максимальное значение (2^15)-1).

В Arduino Due переменные типа *int* - 32-битные (4-байта), что дает возможность хранить значения в диапазоне от -2 147 483 648 до 2 147 483 647 (минимальное значение -2^31 и максимальное значение (2^31)-1).

В переменных типа *int* отрицательные числа представляются с помощью техники дополнительного кода. Старший бит, который иногда называют "знаковым битом", указывает на то, является ли данное число отрицательным. Остальные биты инвертируются, после чего к результату добавляется 1.

Ардуино берет на себя обработку отрицательных чисел, поэтому арифметические операции с ними выглядят так, как вы этого ожидаете. Неожиданные сложности могут возникнуть только при работе с оператором сдвига вправо >>.

Пример:

```
int ledPin = 13;
```

Синтаксис

```
int var = val;
```
var - имя вашей переменной типа int
val - значение, присваиваемое этой переменной

Подсказка

В ситуациях, когда значение переменной стремится превысить свой максимум, оно сбрасывается в минимальное значение, причем данный принцип работает в оба направления. Например, для 16-битной переменной int:

```
int x;
x = -32768;
x = x - 1;        // в x теперь хранится 32767 - произошел сброс
в отрицательном направлении
```

```
x = 32767;
x = x + 1;        // в результате сброса в x теперь хранится -
32768
```

Смотрите также

byte
unsigned int
long
unsigned long
Целочисленные константы
Объявление переменных

unsigned int

Описание

В Uno и других платах на базе микроконтроллеров ATmega, переменные типа unsigned int (беззнаковые целые) схожи с int-переменными тем, что они содержат двухбайтовые значения. Отличие состоит в том, что вместо отрицательных чисел они могут хранить только положительные значения в удобном диапазоне от 0 до 65535 ((2^16)-1).

В Arduino Due такие переменные занимают 4 байта (32 бита), что позволяет хранить значения в диапазоне от 0 до 4 294 967 295 (2^32 - 1).

Разница между беззнаковыми (unsigned int) и знаковыми (int) целыми числами заключается в том, как интерпретируется их старший бит (иногда называемый "знаковым битом"). В Ардуино переменные типа int (знаковые) обрабатываются следующим образом: если старший бит - "1", то число интерпретируется как отрицательное, а остальные 15 бит интерпретируются согласно принципам дополнительного кода.

Пример

```
unsigned int ledPin = 13;
```

Синтаксис

```
unsigned int var = val;
```

var - имя переменной типа unsigned int
val - значение, присваиваемое этой переменной

Подсказка

В ситуациях, когда значение переменной стремится превысить свой максимум, оно сбрасывается в минимальное значение, причем данный принцип работает в оба направления.

```
unsigned int x
x = 0;
x = x - 1;        // x теперь содержит 65535 - произошел сброс в отрицательном направлении
x = x + 1;        // в результате сброса x теперь содержит 0
```

Смотрите также

byte
int

word

Описание

Переменные типа word хранят 16-битное беззнаковое число в диапазоне от 0 до 65535. То же самое, что и unsigned int.

Пример

```
word w = 10000;
```

Смотрите также

byte
word()

long

Описание

Переменные типа long обладают расширенным размером для хранения чисел и имеют размерность 32 бита (4 байта), что позволяет им хранить числа в диапазоне от -2 147 483 648 до 2 147 483 647.

Пример

```
long speedOfLight = 186000L;   // для справки о префиксе 'L' смотрите раздел о целочисленных константах
```

Синтаксис

```
long var = val;
```

var - имя переменной типа long
val - значение, присваиваемое этой переменной

Смотрите также

byte
int
unsigned int
unsigned long
Целочисленные константы
Объявление переменных

unsigned long

Описание

Переменные типа unsigned long обладают расширенным размером для хранения чисел и имеют размерность 32 бита (4 байта). Переменные типа unsigned long, в отличие от обычного long, хранят только положительные числа в диапазоне от 0 до 4 294 967 295 (2^32 - 1).

Пример

```
unsigned long time;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  Serial.print("Time: ");
  time = millis();
  //выводим время с момента запуска программы
  Serial.println(time);
  //ждем 1 секунду, чтобы не отсылать большой массив данных
  delay(1000);
}
```

Синтаксис

```
unsigned long var = val;
```

var - имя переменной типа unsigned long
val - значение, присваиваемое этой переменной

Смотрите также

byte
int
unsigned int
long
Целочисленные константы
Объявление переменных

short

Описание

short - это 16-битный тип данных.

На всех платах Ардуино (как на основе ATmega, так и на основе ARM-микроконтроллеров) переменные типа *short* содержат 16-битные (2-байтовые) значения, что позволяет хранить в них числа в диапазоне от -32768 (-2^15) до 32767 (2^15 - 1).

Пример

```
short ledPin = 13;
```

Синтаксис

```
short var = val;
```

var - имя переменной типа short
val - значение, присваиваемое этой переменной

Смотрите также

byte
int
unsigned int
long
unsigned long
Целочисленные константы
Объявление переменных

float

Описание

Тип данных для чисел с плавающей точкой (чисел с десятичным разделителем). Числа с плавающей точкой часто используются для представления аналоговых или непрерывных величин, поскольку позволяют описать их более точно, чем целые числа. Числа с плавающей точкой представляют собой 32 бита (4 байта) информации и могут достигать огромных значений от -3.4028235E+38 до 3.4028235E+38.

Точность дробных чисел типа float составляет 6-7 десятичных знаков. Здесь имеется ввиду общее количество цифр, а не количество знаков после запятой. В отличие от других платформ, где более высокой точности можно добиться за счет использования типа double (до 15 знаков), в Ардуино тип double имеет такую же размерность, как и float.

Следует иметь ввиду, что числа с плавающей точкой не являются точными, что может приводить к неожиданным результатам при их сравнении. Например, 6.0 / 3.0 может не равняться 2.0. Поэтому, вместо сравнения двух чисел следует проверять, является ли абсолютное значение их разности меньше некоторого небольшого значения.

Помимо этого, математические операции с дробными числами осуществляются гораздо медленнее, чем операции с целыми числами. Поэтому в некоторых ситуациях их следует избегать, например, в циклах, внутри которых осуществляются критичные ко времени функции. С целью повышения производительности программисты часто идут на увеличение программного кода для того, чтобы преобразовать дробные вычисления к целочисленным.

Примеры

```
float myfloat;
float sensorCalbrate = 1.117;
```

Синтаксис

```
float var = val;
```
var - имя переменной типа float
val - значение, присваиваемое этой переменной

Примеры кода

```
int x;
int y;
```

```
float z;

x = 1;
y = x / 2;            // y содержит 0, т.к. целые типы не могут
хранить дробную часть числа
z = (float)x / 2.0;   // z содержит .5 (необходимо использовать
2.0, а не 2)
```

Смотрите также

int
double
Объявление переменных

double

Описание

Дробное число двойной точности. В Uno и других платах на базе микроконтроллеров ATmega переменные типа double занимают 4 байта. Другими словами, в этих устройствах переменные типа double полностью аналогичны переменным float без какого-либо прироста точности.

В Arduino Duo переменные double имеют точность 8 байт (64 бита).

Совет

При переносе кода, содержащего переменные типа double, из других источников следует проверить, соответствует ли подразумеваемая точность фактической точности, которой можно добиться от плат Ардуино на микроконтроллерах ATmega.

Смотрите также

float

string

Описание

Текстовые строки могут быть объявлены двумя способами: можно использовать тип данных String, который входит в ядро, начиная с версии 0019; либо объявить строку как массив символов char с нулевым символом в конце. На этой странице описан второй способ. Для получения более подробной информации об объекте String, предоставляющем больше возможностей ценой большего расхода памяти, см. страницу String - объект.

Примеры

Ниже представлены примеры правильного объявления строк.

```
char Str1[15];
char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};
char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};
char Str4[ ] = "arduino";
char Str5[8] = "arduino";
char Str6[15] = "arduino";
```

Допускаемые операции при объявлении строк

Объявить массив символов без его инициализации (Str1)
Объявить массив символов с одним избыточным элементом, компилятор сам добавит требуемый нулевой символ (Str2)
Добавить нулевой символ явно (Str3)
Инициализировать массив с помощью строковой константы, заключенной в кавычки; компилятор создаст массив необходимого размера с нулевым символом в конце (Str4)
Инициализировать массив с помощью строковой константы, явно указав его размер (Str5)
Инициализировать массив избыточного размера, оставив место для более длинных строк (Str6)

Нулевой завершающий символ

Как правило, все строки завершаются нулевым символом (ASCII код 0), который позволяет функциям (подобным Serial.print()) определять длину строки. Без этого символа они продолжали ли бы последовательно считывать байты памяти, которые фактически уже не являлись бы частью строки.

По сути, это означает, что длина вашей строки должна быть на 1 символ

больше, чем текст, который вы хотели бы в ней хранить. Именно поэтому Str2 и Str5 должны быть длиной 8 символов, несмотря на то, что слово "arduino" занимает всего 7 - последняя позиция автоматически заполняется нулевым символом. Размер Str4 автоматически станет равным 8 - один символ требуется для завершающего нуля. В строке Str3 мы самостоятельно указали нулевой символ (обозначается '\0').

Следует иметь в виду, что в целом можно объявить строку и без завершающего нулевого символа (например, если задать длину Str2 равной 7, а не 8). Однако это приведет к неработоспособности большинства строковых функций, поэтому не следует намеренно так делать. Такая ошибка может быть причиной странного поведения или появления сторонних символов при работе со строками.

Одинарные или двойные кавычки?

Строки всегда объявляются в двойных кавычках ("Abc"), а символы всегда объявляются в одинарных кавычках ('A').

Перенос длинных строк

Длинные строки можно переносить так:

```
char myString[] = "This is the first line"
" this is the second line"
" etcetera";
```

Массивы строк

При работе с большими объемами текста (например, в проектах, работающих с LCD-экраном) часто удобно использовать массивы строк. Поскольку строки сами по себе являются массивами, то фактически, это - пример двумерного массива.

В нижеследующей программе звездочка после указания типа данных char "char*" показывает, что переменная является массивом "указателей". Все имена массивов фактически являются указателями, поэтому звездочка необходима для создания массива массивов. Указатели в C - одна из наиболее сложных вещей для начинающих, но в данном случае глубокого понимания указателей для их эффективного использования вовсе не требуется.

Пример

```
char* myStrings[]={"This is string 1", "This is string 2", "This
is string 3",
"This is string 4", "This is string 5","This is string 6"};
```

```
void setup(){
Serial.begin(9600);
}

void loop(){
for (int i = 0; i < 6; i++){
   Serial.println(myStrings[i]);
   delay(500);
   }
}
```

Смотрите также

array
PROGMEM
Объявление переменных

Массивы

Массив - это набор переменных, доступ к которым осуществляется по индексу. Массивы в языке программирования С, использующемся при программировании Ардуино, могут представлять собой сложные структуры, но для понимания использование обычных одномерных массивов проще.

Создание (объявление) массива

Ниже представлены правильные способы создания (объявления) массивов.

```
int myInts[6];
int myPins[] = {2, 4, 8, 3, 6};
int mySensVals[6] = {2, 4, -8, 3, 2};
char message[6] = "hello";
```

Можно объявить массив без его инициализации, как myInts.

В myPins мы объявили массив без прямого указания его размера. Компилятор сам посчитает элементы и создаст массив соответствующего размера.

В конце концов, можно и проинициализировать массив, и указать его размер, как mySensVals. Следует помнить, что при объявлении массива типа char, в нем необходимо место для хранения обязательного нулевого символа, поэтому размер массива должен быть на один символ больше, чем требует инициализируемое значение.

Обращение к элементам массива

Нумерация элементов массива **начинается с нуля**, т.е. первый элемент массива имеет индекс 0. Следовательно, применительно к проинициализированным выше массивам,

```
mySensVals[0] == 2, mySensVals[1] == 4, и т.д.
```

Это также означает, что в массиве из 10 элементов, последний элемент имеет индекс 9. Следовательно:

```
int myArray[10]={9,3,2,4,3,2,7,8,9,11};
     // myArray[9]    содержит 11
     // myArray[10]   ошибочный результат, который может
содержать случайные значения (из-за другого адреса памяти)
```

Поэтому, необходимо быть внимательным при обращении к массивам. Обращение к элементу за пределами массива (когда указанный индекс больше, чем объявленный размер массива - 1) приведет к чтению данных из ячейки памяти, использующейся для других целей. Считывание из этой области, вероятно, не приведет ни к чему, кроме получения неверных

данных. Запись же в случайные области памяти - определенно плохая идея, и часто может приводить к не желаемым результатам, таким, как зависания или сбои в работе программы. Кроме того, такие ошибки трудно отыскать.

В отличие от BASIC или JAVA, компилятор C не проверяет правильность индексов при обращении к элементам массива.

Как записать значение в массив:

```
mySensVals[0] = 10;;
```

Как считать значение из массива:

```
x = mySensVals[4];
```

Массивы и циклы FOR

Работа с массивами часто осуществляется внутри циклов FOR, в которых счетчик цикла используется в качестве индекса каждого элемента массива. Например, программа вывода элементов массива через последовательный порт может выглядеть так:

```
int i;
for (i = 0; i < 5; i = i + 1) {
  Serial.println(myPins[i]);
}
```

Пример

Полную версию программы, демонстрирующую работу с массивами, смотрите в примере Knight Rider из раздела "Примеры".

Смотрите также

Объявление переменных
PROGMEM

String

Описание

Класс String появился в ядре, начиная с версии 0019. Он позволяет осуществлять более сложную обработку текстовых строк, чем обычные массивы символов. С помощью класса String вы можете объединять и расширять строки, осуществлять поиск и замену подстрок, и многое другое. Он занимает больше памяти, чем простой массив символов, но предоставляет больше возможностей.

Общепринято называть строки из массивов символов с маленькой литеры "s" (string), а экземпляры класса String - с большой "S". Следует иметь ввиду, что строковые константы, заключенные в двойные кавычки, интерпретируются как массивы символов, а не экземпляры класса String.

String()

Описание

Создает экземпляр класса String. Существует несколько разных версий этой функции, которые создают объект String из разных типов данных (другими словами, формируют строку из последовательности символов):

строковая константа в двойных кавычках (т.е. массив символов)
одиночный символ в одинарных кавычках
другой экземпляр класса String
целочисленная константа типа int или long
целочисленная константа типа int или long с указанием основания системы счисления
целочисленная переменная типа int или long
целочисленная переменная типа int или long с указанием основания системы счисления

При создании объекта String из числа, результирующая строка будет содержать ASCII-представление этого числа. По умолчанию считается, что число указано в десятичной системе, поэтому:

```
String thisString = String(13)
```

приведет к созданию строки "13". Можно указывать основание системы счисления, например:

```
String thisString = String(13, HEX)
```

приведет к созданию строки "D", что является шестнадцатиричным представлением десятичного числа 13. То же самое в двоичной системе:

```
String thisString = String(13, BIN)
```

приведет к созданию строки "1101", что является двоичным представлением числа 13.

Синтаксис

```
String(val)
String(val, base)
```

Параметры

val: переменная, значение которой необходимо представить в виде объекта String - *string, char, byte, int, long, unsigned int, unsigned long*
base (не обязательный параметра) - основание системы счисления, в котором необходимо представить целое число

Возвращаемые значения

нет

Пример

Ниже перечислены примеры корректного объявления строк String.

```
String stringOne = "Hello String";                       //
использование строковой константы
String stringOne =  String('a');                         //
преобразование символа в String
String stringTwo =  String("This is a string");          //
преобразование строковой константы в объект String
String stringOne =  String(stringTwo + " with more");    //
конкатенация двух строк
String stringOne =  String(13);                          //
использование целочисленной константы
String stringOne =  String(analogRead(0), DEC);          //
использование int с основанием системы счисления
String stringOne =  String(45, HEX);                     //
использование int с основанием 16
String stringOne =  String(255, BIN);                    //
использование int с основанием 2
String stringOne =  String(millis(), DEC);               //
использование long с основанием системы счисления
```

Смотрите также

[String Constructor Tutorial]

charAt()

Описание

Возвращает указанный символ из строки String.

Синтаксис

```
string.charAt(n)
```

Параметры

string: переменная String
n: номера символа

Возвращаемые значения

n-ный символ строки String

Смотрите также

setCharAt()
[]

compareTo()

Описание

Проверяет две строки типа String на равенство, а также позволяет определить, какая из сравниваем строк идет раньше другой в алфавитном порядке. Строки сравниваются посимвольно по ASCII-коду каждого символа. Например, символ 'a' идет до символа 'b', но после символа 'A'. Все цифры следуют до букв.

Синтаксис

```
string.compareTo(string2)
```

Параметры

string: переменная типа String

string2: вторая переменная типа String

Возвращаемые значения

отрицательное число: если строка string идет до строки string2 в алфавитном порядке

0: если строка string эквивалентна строке string2

положительное число: если string идет после string2

Пример

StringComparisonOperators

Смотрите также

substring()

concat()

Описание

Объединяет две строки в одну строку (операция конкатенации). Вторая строка добавляется к первой, а результат сложения помещается в новый объект String.

Синтаксис

```
string.concat(string, string2)
```

Параметры

string, string2: переменные типа String

Возвращаемые значения

новый объект String, содержащий комбинацию двух строк.

Пример

StringAppendOperator

Смотрите также

String Addition operator

endsWith()

Описание

Проверяет, завершается ли строка одним из символов, содержащихся во второй строке (String).

Синтаксис

```
string.endsWith(string2)
```

Параметры

string: переменная типа String

string2: вторая переменная типа String

Возвращаемые значения

true: если первая строка (string) завершается символом, содержащимся в строке string2

false: в противном случае

Пример

StringStartsWithEndsWith

Смотрите также

startsWith()
lastIndexOf()
indexOf()

equalsIgnoreCase()

Описание

Проверяет две строки на равенство. Функция сравнения не чувствительна к регистру символов, т.е. String("hello") считается эквивалентной строке String ("HELLO").

Синтаксис

```
string.equalsIgnoreCase(string2)
```

Параметры

string, string2: переменные типа String

Возвращаемые значения

true: если строка string эквивалентна строке string2 (без учета регистра символов)

false: в противном случае

Пример

StringComparisonOperators

Смотрите также

equals()
compareTo()

equals()

Описание

Проверяет две строки на равенство. Функция сравнения чувствительная к регистру символов, т.е. строка "hello" не считается эквивалентной строке "HELLO".

Синтаксис

```
string.equals(string2)
```

Параметры

string, string2: переменные типа String

Возвращаемые значения

true: если строка string эквивалентна строке string2

false: в противном случае

Пример

StringComparisonOperators

Смотрите также

equalsIgnoreCase()
compareTo()

indexOf()

Описание

Осуществляет поиск символа или подстроки в строке (String). По умолчанию, поиск осуществляется с начала строки, однако можно указать и определенную позицию, с которой необходимо начать поиск. Такая возможность позволяет находить все вхождения подстроки в строке.

Синтаксис

```
string.indexOf(val)
string.indexOf(val, from)
```

Параметры

string: переменная типа String

val: искомое значение - *char* или *String*

from: начальная позиция для поиска

Возвращаемые значения

Индекс подстроки val в строке String, или -1, если подстрока не найдена.

Пример

StringIndexOf

Смотрите также

lastIndexOf()
startsWith()
endsWith()

lastIndexOf()

Описание

Осуществляет поиск символа или подстроки в строке (String). По умолчанию, поиск осуществляется с конца строки, однако можно указать и определенную позицию, с которой необходимо просматривать строку в обратном порядке. Такая возможность позволяет находить все вхождения подстроки в строке.

Синтаксис

```
string.lastIndexOf(val)
string.lastIndexOf(val, from)
```

Параметры

string: переменная типа String

val: искомое значение - *char* или *String*

from: начальная позиция для поиска в обратном порядке

Возвращаемые значения

Индекс подстроки val в строке String, или -1, если подстрока не найдена.

Пример

StringIndexOf

Смотрите также

indexOf()
startsWith()
endsWith()

length()

Описание

Возвращает длину строки String в символах. (Обратите внимание, что при подсчете количества символов функция не учитывает завершающий нулевой символ).

Синтаксис

```
string.length()
```

Параметры

string: переменная типа String

Возвращаемые значения

Длина строки String в символах

Пример

StringLengthTrim

getBytes()

Описание

Копирует символы строки в указанные буфер.

Синтаксис

```
string.getBytes(buf, len)
```

Параметры

string: переменная типа String

buf: буфер для копирования символов (*byte []*)

len: размер буфера (*unsigned int*)

Возвращаемые значения

нет

Смотрите также

toCharArray()

replace()

Описание

Функция String.replace() позволяет заменить в строке все вхождения определенного символа на другой символ. Эту функцию можно также использовать для замены подстроки в строке.

Синтаксис

```
string.replace(substring1, substring2)
```

Параметры

string: переменная типа String

substring1: еще одна переменная типа String

substring2: еще одна переменная типа String

Возвращаемые значения

Объект String, содержащий результирующую строку.

Пример

StringReplace

Смотрите также

substring()
toLowerCase()
toUpperCase()

reserve()

Описание

Функция String.reserve() позволяет выделить в памяти буфер для работы со строками.

Синтаксис

```
string.reserve(size)
```

Параметры

size: количество байт, которое необходимо зарезервировать в памяти для обработки строк, *unsigned int*

Возвращаемые значения

нет

Пример

```
String myString;

void setup() {
  // инициализируем последовательный интерфейс и ожидаем
открытия порта:
  Serial.begin(9600);
  while (!Serial) {
    ; // ожидаем подключения к последовательному порту. Нужно
только для Leonardo
  }

  myString.reserve(26);
  myString = "i=";
  myString += "1234";
  myString += ", is that ok?";

  // выводим строку:
  Serial.println(myString);
}

void loop() {
 // ничего не делаем
```

```
}
```

setCharAt()

Описание

Позволяет изменить определенный символ в строке. При указание индекса, превышающего размерность строки, функция ничего не делает.

Синтаксис

```
string.setCharAt(index, c)
```

Параметры

string: переменная типа String

index: индекс символа, который необходимо изменить

c: символ, который необходимо записать в указанную позицию строки

Возвращаемые значения

нет

Смотрите также

charAt()

startsWith()

Описание

Проверяет, начинается ли строка одним из символов, содержащихся во второй строке (String).

Синтаксис

```
string.startsWith(string2)
```

Параметры

string, string2: переменные типа String

Возвращаемые значения

true: если первая строка (string) начинается символом, содержащимя в строке string2

false: в противном случае

Пример

StringStartsWithEndsWith

Смотрите также

endsWith()
lastIndexOf()
indexOf()

substring()

Описание

Возвращает подстроку в строке (String). Функция принимает два параметра: начальный индекс и конечный индекс в строке (необязательный параметр). При этом начальный индекс является включительным (соответствующий символ будет включен в подстроку), а конечный индекс - не включительным (соответствующий символ не включается в результирующую подстроку). Если конечный индекс отсутствует, то возвращаемое функцией значение будет содержать все символы от начального индекса и до конца строки.

Синтаксис

```
string.substring(from)
string.substring(from, to)
```

Параметры

string: переменная типа String

from: начальный индекс в строке

to (необязательный параметр): конечный индекс в строке

Возвращаемые значения

подстрока

Пример

StringSubstring

toCharArray()

Описание

Копирует символы строки в указанный буфер.

Синтаксис

```
string.toCharArray(buf, len)
```

Параметры

string: переменная типа String

buf: буфер, в который необходимо скопировать символы строки (*char []*)

len: размер буфера (*unsigned int*)

Возвращаемые значения

нет

Смотрите также

getBytes()

toInt()

Описание

Преобразовывает строку (String) в целое число. Строка должна начинаться с символьной записи целого числа. Если же строка содержит не целочисленное значение, функция прекратит преобразование.

Синтаксис

```
string.toInt()
```

Параметры

string: переменная типа String

Возвращаемые значения

long

Если в строке содержится не числовое значение и преобразование не может завершиться, функция возвращает 0.

Пример

StringToIntExample

Смотрите также

compareTo()

toLowerCase()

Описание

Возвращает строку, записанную в нижнем регистре символов. Начиная с версии Ардуино 1.0, функция toLowerCase() преобразовывает существующую строку, а не возвращает новую.

Синтаксис

```
string.toLowerCase()
```

Параметры

string: переменная типа String

Возвращаемые значения

нет

Пример

StringCaseChanges

Смотрите также

toUpperCase()

toUpperCase()

Описание

Возвращает строку, записанную в верхнем регистре символов. Начиная с версии Ардуино 1.0, функция toUpperCase() преобразовывает существующую строку, а не возвращает новую.

Синтаксис

```
string.toUpperCase())
```

Параметры

string: переменная типа String

Возвращаемые значения

нет

Пример

StringCaseChanges

Смотрите также

toLowerCase()

trim()

Описание

Функция обрезает все пробелы в начале и конце указанной строки. Начиная с версии Ардуино 1.0, функция trim() преобразовывает существующую строку, а не возвращает новую.

Синтаксис

```
string.trim()
```

Параметры

string: переменная типа String

Возвращаемые значения

нет

Пример

StringLengthTrim

[] (доступ к элементу)

Описание

Позволяет обратиться к определенному символу строки.

Синтаксис

```
char thisChar = string1[n]
```

Параметры

char thisChar - символьная переменная
string1 - строковая переменная
int n - числовая переменная

Возвращаемые значения

n-ный символ строки. То же самое, что и charAt().

Пример

StringCharacters

Смотрите также

StringCharAt

+ (конкатенация)

Описание

Объединяет две строки в одну строку (операция конкатенации). Вторая строка добавляется к первой, а результат сложения помещается в новый объект String. Работает аналогично функции string.concat()

Синтаксис

```
string3 = string1 + string 2; string3 += string2;
```

Параметры

string, string2, string3: переменные типа String

Возвращаемые значения

новый объект String, содержащий комбинацию складываемых строк.

Пример

StringAppendOperator

Смотрите также

StringAdditionOperator

== (сравнение)

Описание

Проверяет две строки на равенство. Операция сравнения чувствительная к регистру символов, т.е. строка "hello" не считается эквивалентной строке "HELLO". Работает аналогично функции string.equals()

Синтаксис

```
string1 == string2
```

Параметры

string1, string2: переменные типа String

Возвращаемые значения

true: если строка string1 эквивалентна строке string2

false: в противном случае

Пример

StringComparisonOperators

Смотрите также

equals()
equalsIgnoreCase()
compareTo()

char()

Описание

Приводит значение к типу char.

Синтаксис

```
char(x)
```

Параметры

x: значение любого типа

Возвращаемые значения

char

Смотрите также

char

byte()

Описание

Приводит значение к типу byte.

Синтаксис

```
byte(x)
```

Параметры

x: значение любого типа

Возвращаемые значения

byte

Смотрите также

byte

int()

Описание

Приводит значение к типу int.

Синтаксис

```
int(x)
```

Параметры

x: значение любого типа

Возвращаемые значения

int

Смотрите также

int

word()

Описание

Приводит значение к типу word или создает значение типа word из двух байт.

Синтаксис

```
word(x)
word(h, l)
```

Параметры

x: значение любого типа

h: старший байт (левая часть) значения word

l: младший байт (правая часть) значения word

Возвращаемые значения

word

Смотрите также

word

long()

Описание

Приводит значение к типу long.

Синтаксис

```
long(x)
```

Параметры

x: значение любого типа

Возвращаемые значения

long

Смотрите также

long

float()

Описание

Приводит значение к типу float.

Синтаксис

```
float(x)
```

Параметры

x: значение любого типа

Возвращаемые значения

float

Смотрите также

float

Область видимости переменной

В языке программирования С, использующемся при программировании Ардуино, переменные имеют свойство, называемое *область видимости*, чего нельзя сказать про первые языки программирования (подобные BASIC), в которых все переменные являются *глобальными*.

Глобальная переменная - эта та переменная, которая может быть доступна ("*видна*") из любой функции программы. Локальные переменные доступны только внутри тех функций, в которых они объявлены. При программировании Ардуино, любая переменная, объявленная за пределами функции (таких как, setup(), loop(), и т.д.), является глобальной переменной.

По мере роста программ и увеличения их сложности, локальные переменные становятся незаменимым инструментом, гарантирующим, что доступ к переменным будет иметь только та функция, в которой они объявлены. Это предотвращает ошибки в программе, при которых одна функция случайно изменяет переменные, используемые в другой функции.

Также иногда удобно объявить и инициализировать переменную внутри цикла *for*. В этом случае переменная будет доступна только в пределах скобок цикла for.

Пример:

```
int gPWMval;  // эта переменная будет доступна из любой функции

void setup()
{
  // ...
}

void loop()
{
  int i;    // переменная "i" "видна" только внутри "loop"
  float f;  // переменная "f" "видна" только внутри "loop"
  // ...

  for (int j = 0; j <100; j++){
  // переменная j доступна только внутри скобок цикла for
  }

}
```

Static

Ключевое слово *static* используется для создания переменных, которые будут видны только одной функции. Однако, в отличие от локальных переменных, которые создаются и уничтожаются при каждом вызове функции, переменные *static* сохраняют свое значение между вызовами.

Переменные, объявленные как *static*, создаются и инициализируются только при первом вызове функции.

Пример:

```
/* RandomWalk
* Paul Badger 2007
* RandomWalk в случайном порядке перемещается вверх или вниз
между двумя
* точками. Длина максимального перемещения за один цикл задается
* параметром "stepsize".
* Статическая переменная увеличивается или уменьшается на
случайную величину.
* Эта техника также известна как "розовый шум" или "пьяная
походка".
*/

#define randomWalkLowRange -20
#define randomWalkHighRange 20
int stepsize;

int thisTime;
int total;

void setup()
{
  Serial.begin(9600);
}

void loop()
{         //  функция randomWalk
  stepsize = 5;
  thisTime = randomWalk(stepsize);
  Serial.println(thisTime);
   delay(10);
}

int randomWalk(int moveSize){
  static int place;      // переменная для хранения величины
```

```
случайного перемещения - объявлена как static, поэтому
                         // сохраняет свое значение между
вызовами функции. При этом другие функции не могут ее изменить

  place = place + (random(-moveSize, moveSize + 1));

  if (place < randomWalkLowRange){                    // проверка
нижнего и верхнего пределов
    place = place + (randomWalkLowRange - place);     //
восстановление числа в положительном направлении
  }
  else if(place > randomWalkHighRange){
    place = place - (place - randomWalkHighRange);     //
восстановление числа в отрицательном направлении
  }

  return place;
}
```

Ключевое слово volatile

volatile - это ключевое слово, известное как *спецификатор* переменной. Как правило, употребляется перед указанием типа переменной, чтобы изменить порядок ее обработки компилятором и последующей программой.

Объявление переменной как volatile - это директива компилятору. Компилятор - программа, которая переводит программный код C/C++ в машинный код, представляющий собой набор команд для микроконтроллера ATmega в Ардуино.

По сути, эта директива заставляет компилятор размещать переменную в ОЗУ, а не во внутренних регистрах, использующихся для временного хранения и обработки различных переменных. При определенных условиях, значение переменной, хранимой в регистрах, может быть неточным.

Переменную необходимо объявлять как *volatile* в тех случаях, когда ее значение может быть изменено чем-либо, не зависящем от того участка кода, в котором она фигурирует (например, параллельно выполняющимся потоком). Применительно к Ардуино, единственное место, где подобное может случиться - это участки кода, связанные с прерываниями (также называемые процедурами обработки прерываний).

Пример:

```
// переключение светодиода при изменении состояния вывода

int pin = 13;
volatile int state = LOW;

void setup()
{
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);
}

void loop()
{
  digitalWrite(pin, state);
}

void blink()
{
  state = !state;
}
```

Ключевое слово const

Ключевое слово **const** обозначает константу. Это *спецификатор*, который изменяет поведение переменной и делает ее доступной только для чтения. Другими словами, эта переменная может использоваться так же, как и любая другая переменная этого же типа, однако ее значение изменить нельзя. Если вы попытаетесь переприсвоить значение константе, компилятор выдаст ошибку.

На константы, обозначенные ключевым словом *const*, распространяются те же правила области видимости, как и для обычных переменных. Именно поэтому использование ключевого слова *keyword* при объявлении констант более предпочтительно, чем использование директивы *#define*.

Пример

```
const float pi = 3.14;
float x;

// ....

x = pi * 2;     // удобно использовать константы при
математических вычислениях

pi = 7;         // ошибка - нельзя записывать значения (изменять)
константы
```

#define или const

Для создания числовых или строковых констант можно использовать как **const**, так и **#define**. Для массивов необходимо использовать **const**. В общем случае при объявлении констант предпочтительнее использовать *const* вместо *#define*.

Смотрите также

#define
volatile

sizeof

Описание

Оператор sizeof возвращает количество байт, занимаемых типом переменной, либо количество байт, занимаемых массивом.

Синтаксис

```
sizeof(variable)
```

Параметры

variable: переменная любого типа или массив (например, int, float, byte)

Пример кода

Оператор sizeof удобно использовать при работе с массивами, особенно в тех случаях, когда размерность массива заранее неизвестна или может меняться.

Следующая программа посимвольно выводит строку. Если изменить исходную фразу - программа останется работоспособной, независимо от длины текста.

```
char myStr[] = "this is a test";
int i;

void setup(){
  Serial.begin(9600);
}

void loop() {
  for (i = 0; i < sizeof(myStr) - 1; i++){
    Serial.print(i, DEC);
    Serial.print(" = ");
    Serial.write(myStr[i]);
    Serial.println();
  }
  delay(5000); // задержка программы
}
```

Обратите внимание, что sizeof возвращает общее количество байтов. Поэтому, при работе с массивами более объемных типов данных (такими, как int), цикл будет выглядеть примерно так. Кроме того, не забывайте, что правильно объявленная строка заканчивается нулевым символом с ASCII-кодом 0.

```
for (i = 0; i < (sizeof(myInts)/sizeof(int)) - 1; i++) {
  // какие-либо операции с myInts[i]
}
```

Библиотека EEPROM

В микроконтроллере Ардуино есть EEPROM - память, в которой информация сохраняется даже после выключения устройства (подобно маленькому жесткому диску). Данная библиотека позволяет записывать и считывать информацию из этой памяти.

Объем памяти EEPROM различных микроконтроллеров, входящих в состав Ардуино, может отличаться: 1024 байта в ATmega328, 512 байт - в ATmega168 и ATmega8, 4 КБ (4096 байт) - в ATmega1280 и ATmega2560.

Функции

read()
write()

read()

Описание

Считывает байт из EEPROM. Те байты, которые никогда не подвергались записи, имеют значение 255.

Синтаксис

```
EEPROM.read(address)
```

Параметры

address: адрес байта, значение которого необходимо считать (*int*)

Возвращаемые значения

значение указанного байта (*byte*)

Пример

```
#include <eeprom.h>

int a = 0;
int value;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  value = EEPROM.read(a);

  Serial.print(a);
  Serial.print("\t");
  Serial.print(value);
  Serial.println();

  a = a + 1;

  if (a == 512)
    a = 0;
```

```
  delay(500);
}</eeprom.h>
```

Смотрите также

write()

write()

Описание

Записывает байт в EEPROM.

Синтаксис

```
EEPROM.write(address, value)
```

Параметры

address: адрес байта (нумерация с 0), значение которого необходимо записать (*int*).

value: записываемое значение, от 0 до 255 (*byte*)

Возвращаемые значения

нет

Примечание

Процедура записи в EEPROM занимает 3.3 мс. При частой записи следует помнить, что память EEPROM имеет ограниченное количество циклов записи/чтения (100 000).

Пример

```
#include <eeprom.h>

void setup()
{
  for (int i = 0; i < 512; i++)
    EEPROM.write(i, i);
}

void loop()
{
}</eeprom.h>
```

Смотрите также

read()

Библиотека SD

Библиотека SD позволяет считывать и записывать информацию на SD-карту памяти (например, на плате расширения Arduino Ethernet). Она основана на библиотеке sdfatlib (автор William Greiman). Библиотека поддерживает работу со стандартными картами памяти типа SD и SDHC, отформатированными в файловой системе FAT16 или FAT32. При работе с картой памяти необходимо использовать короткие имена файлов в формате 8.3 (8 символов - имя файла, 3 символа - расширение). Функции библиотеки SD в качестве параметра могут принимать не только имя файла, но и путь к нему. При этом в качестве разделителя между каталогами используется прямой слеш (например, "directory/filename.txt"). Добавление косой черты перед именем файла необязательно, поскольку рабочей директорией всегда является корневой каталог карты памяти (таким образом, имя "/file.txt" эквивалентно "file.txt"). Начиная с версии 1.0, в библиотеке реализована возможность одновременного открытия нескольких файлов.

Взаимодействие между микроконтроллером и SD-картой памяти осуществляется по шине SPI, объединяющей в себе выводы 11, 12 и 13 (на большинстве плат Ардуино), либо 50, 51 и 52 (на Arduino Mega). Помимо перечисленных, еще один вывод должен использоваться для активизации SD-карты. Для этого может использоваться как аппаратный вывод SS - вывод 10 (на большинстве плат Ардуино) или вывод 53 (на Arduino Mega), так и любой другой вывод, указанный при вызове метода SD.begin(). **Обратите внимание, что для корректной работы библиотеки SD, вывод SS должен быть сконфигурирован как выход, даже в тех случаях, когда он не используется.**

Примеры

Datalogger: Запись данных с трех аналоговых датчиков на SD-карту памяти с помощью библиотеки SD
DumpFile: Считывание файла с SD-карты памяти и отправка его содержимого через последовательный порт
Files: Создание и удаление файла с SD-карты памяти
ReadWrite: Чтение и запись данных в файл на SD-карте
CardInfo: Получение информации об SD-карте памяти

Класс SD содержит функции для доступа к SD-карте памяти и позволяет совершать различные операции над файлами и каталогами.

begin()

Описание

Инициализирует библиотеку и SD-карту памяти. Метод активизирует шину SPI (цифровые выводы 11, 12 и 13 - на большинстве плат Ардуино; 50, 51 и 52 - на Arduino Mega), а также вывод CS, в качестве которого по умолчанию выступает линия SS (на большинстве плат Ардуино - вывод 10, на Arduino Mega - 53). Следует помнить, что для корректной работы интерфейса SPI, аппаратный вывод SS должен быть всегда сконфигурирован как выход, даже если он не используется для активизации SD-карты.

Синтаксис

```
SD.begin()
SD.begin(cspin)
```

Параметры

cspin (*не обязательный параметр*): номер вывода, подключенного к линии CS SD-карты памяти; по умолчанию равен номеру вывода SS аппаратной шины SPI.

Возвращаемые значения

При успешном выполнении функция возвращает true, в противном случае - false.

exists()

Описание

Проверяет существование на SD-карте указанного файла или директории.

Синтаксис

```
SD.exists(filename)
```

Параметры

filename: имя файла, существование которого необходимо проверить. Имя может включать в себя директории, разделенные прямым слешем "/".

Возвращаемые значения

Если указанный файл или директория существует - функция возвращает true, в противном случае - false.

mkdir()

Описание

Создает на SD-карте указанную директорию. Если путь к указанной директории содержит несуществующие папки - функция также их создает. Например, команда SD.mkdir("a/b/c") приведет к созданию трех папок: a, b и c.

Синтаксис

```
SD.mkdir(filename)
```

Параметры

filename: имя создаваемой директории. Имя может включать в себя вложенные папки, разделенные прямым слешем "/".

Возвращаемые значения

Если директория успешно создана - функция возвращает true, в противном случае - false.

Смотрите также

rmdir()

open()

Описание

Функция открывает файл на SD-карте памяти. Если указанный файл не существует, то при открытии для записи он будет создан автоматически. Директория, содержащая файл, заведомо должна существовать.

Примечание: **в один момент времени может быть открыт только один файл.**

Синтаксис

```
SD.open(filepath)
SD.open(filepath, mode)
```

Параметры

filename: имя открываемого файла. Может содержать директории, разделенные прямым слешем "/" - *char **

mode (*не обязательный параметр*): режим открытия файла, по умолчанию - FILE_READ - *byte*. Может принимать одно из следующих значений:

FILE_READ: открыть для чтения с начала файла.
FILE_WRITE: открыть для чтения или записи в конец файла..

Возвращаемые значения

Объект File, ссылающийся на открытый файл; если файл открыть не удалось, возвращаемый объект будет эквивалентен значению false (в логическом смысле). Т.е. значение, возвращаемое этой функцией, можно анализировать с помощью условного оператора if.

Смотрите также

File: close()

remove()

Описание

Функция удаляет файл с SD-карты.

Синтаксис

```
SD.remove(filename)
```

Параметры

filename: имя удаляемого файла. Может содержать директории, разделенные прямым слешем "/".

Возвращаемые значения

Если файл был успешно удален - функция вернет true, в противном случае - false. (Если указанный файл не существовал, возвращаемое значение может быть неопределенным).

Смотрите также

rmdir()

rmdir()

Описание

Удаляет директорию с SD-карты. Указываемая директория должна быть пустой.

Синтаксис

```
SD.rmdir(filename)
```

Параметры

filename: имя удаляемой директории. Может содержать вложенные папки, разделенные прямым слешем "/".

Возвращаемые значения

Если удаление директории прошло успешно - функция вернет true, в противном случае - false. (Если указанная директория не существовала, значение, возвращаемое функцией, может быть неопределенным).

Смотрите также

mkdir()

Этот класс предназначен для чтения и записи данных в отдельные файлы на SD-карте памяти.

available()
close()
flush()
peek()
position()
print()
println()
seek()
size()
read()
write()
isDirectory()
openNextFile()
rewindDirectory()

available()

Описание

Функция проверяет наличие в файле байт, доступных для чтения.

Функция available() является наследником вспомогательного класса Stream.

Синтаксис

```
file.available()
```

Параметры

file: экземпляр класса FIle (возвращаемый функцией SD.open()).

Возвращаемые значения

Количество доступных для чтения байт (*int*).

Смотрите также

read()
peek()
Stream.available()

close()

Описание

Функция закрывает файл, удостоверившись, что все записанные в него данные физически сохранены на SD-карту памяти.

Синтаксис

```
file.close()
```

Параметры

file: экземпляр класса FIle (возвращаемый функцией SD.open()).

Возвращаемые значения

нет

Смотрите также

SD: open()

flush()

Описание

Функция flush() позволяет убедиться, что все записанные в файл данные физически сохранены на SD-карту памяти. Эта же операция выполняется автоматически при закрытии файла.

Синтаксис

```
file.flush()
```

Параметры

file: экземпляр класса FIle (возвращаемый функцией SD.open()).

Возвращаемые значения

нет

Смотрите также

close()

peek()

Описание

Считывает из файла байт данных, не перемещая указатель текущей позиции файла на следующий символ. Т.е. при многократном вызове peek() функция будет возвращать одно и то же значение.

Функция peek() является наследником вспомогательного класса Stream.

Синтаксис

```
file.peek()
```

Параметры

file: экземпляр класса FIle (возвращаемый функцией SD.open()).

Возвращаемые значения

Байт данных (или символ), либо -1, если таковой отсутствует.

Смотрите также

available()
read()
write()
Stream.peek()

position()

Описание

Функция позволяет узнать текущее положение указателя внутри открытого файла (т.е. позицию, с которой будет осуществляться чтение или запись следующего байта).

Синтаксис

```
file.position()
```

Параметры

file: экземпляр класса FIle (возвращаемый функцией SD.open()).

Возвращаемые значения

Номер текущей позиции указателя внутри файла (*unsigned long*)

Смотрите также

seek()

print()

Описание

Функция выводит данные в файл, предварительно открытый для записи. При этом числа выводятся как последовательность цифр, каждая из которых является ASCII-символом (например, число 123 будет отправлено как последовательность из трех символов '1', '2', '3').

Синтаксис

```
file.print(data)
file.print(data, BASE)
```

Параметры

file: экземпляр класса FIle (возвращаемый функцией SD.open()).

data: данные, которые необходимо вывести (char, byte, int, long или string)

BASE (не обязательный параметр): система счисления, в которой необходимо выводить числа: BIN для двоичной системы (основание 2), DEC - для десятичной (основание 10), OCT - для восьмеричной (основание 8), HEX - для шестнадцатеричной (основание 16).

Возвращаемые значения

byte

print() возвращает количество записанных байт. Считывать это значение не обязательно.

Смотрите также

println()
write()

println()

Описание

Функция выводит в файл данные, заканчивающиеся символом перевода каретки и пустой строкой. Перед вызовом функции файл должен быть предварительно открыт для записи. Функция выводит числа в виде последовательности цифр, каждая из которых представляет собой ASCII-символ (например, число 123 будет отправлено как последовательность из трех символов '1', '2', '3').

Синтаксис

```
file.println()
file.println(data)
file.print(data, BASE)
```

Параметры

file: экземпляр класса FIle (возвращаемый функцией SD.open()).

data (не обязательный параметр): данные, которые необходимо вывести (char, byte, int, long или string)

BASE (не обязательный параметр): система счисления, в которой необходимо выводить числа: BIN для двоичной системы (основание 2), DEC - для десятичной (основание 10), OCT - для восьмеричной (основание 8), HEX - для шестнадцатеричной (основание 16).

Возвращаемые значения

byte

println() возвращает количество записанных байт. Считывать это значение не обязательно.

Смотрите также

print()
write()

seek()

Описание

Функция изменяет положение текущей позиции внутри файла. Задаваемая позиция должна лежать в пределах от 0 до текущего размера файла (включительно).

Синтаксис

```
file.seek(pos)
```

Параметры

file: экземпляр класса FIle (возвращаемый функцией SD.open()).

pos: позиция, в которую необходимо переместить указатель (*unsigned long*)

Возвращаемые значения

В случае успешного выполнения функция возвращает true, в противном случае - false (*boolean*).

Смотрите также

position()

size()

Описание

Возвращает размер файла.

Синтаксис

```
file.size()
```

Параметры

file: экземпляр класса FIle (возвращаемый функцией SD.open()).

Возвращаемые значения

Размер файла в байтах (*unsigned long*).

read()

Описание

Функция считывает байт данных из открытого файла.

Функция read() является наследником вспомогательного класса Stream.

Синтаксис

```
file.read()
```

Параметры

file: экземпляр класса FIle (возвращаемый функцией SD.open()).

Возвращаемые значения

Байт данных (или символ), либо -1, если таковых нет.

Смотрите также

available()
peek()
write()
Stream.read()

write()

Описание

Записывает данные в файл.

Синтаксис

```
file.write(data)
file.write(buf, len)
```

Параметры

file: экземпляр класса File (возвращаемый функцией SD.open())

data: записываемый байт данных (byte), символ (char) или строка (char *)

buf: массив символов или байт

len: количество элементов массива

Возвращаемые значения

byte

Функция write() возвращает количество записанных байт. Считывать этот параметр не обязательно.

Смотрите также

print()
println()
read()

isDirectory()

Описание

Директории (или папки) технически являются особым видом файлов. Данная функция позволяет проверить, является ли указанный файл директорией или нет.

Синтаксис

```
file.isDirectory()
```

Параметры

file: экземпляр класса File (возвращаемый функцией file.open())

Возвращаемые значения

boolean

Пример

```
#include <SD.h>

File root;

void setup()
{
  Serial.begin(9600);
  pinMode(10, OUTPUT);

  SD.begin(10);

  root = SD.open("/");

  printDirectory(root, 0);

  Serial.println("done!");
}

void loop()
{
  // после первоначальной настройки ничего не происходит.
}
```

```
void printDirectory(File dir, int numTabs) {
   while(true) {

     File entry =  dir.openNextFile();
     if (! entry) {
       // больше нет файлов
       //Serial.println("**nomorefiles**");
       break;
     }
     for (uint8_t i=0; i<numTabs; i++) {
       Serial.print('\t');
     }
     Serial.print(entry.name());
     if (entry.isDirectory()) {
       Serial.println("/");
       printDirectory(entry, numTabs+1);
     } else {
       // у файлов есть размер, в отличие от директорий
       Serial.print("\t\t");
       Serial.println(entry.size(), DEC);
     }
   }
}
```

Смотрите также

openNextFile()
rewindDirectory()

openNextFile()

Описание

Возвращает имя следующего файла или папки в пределах директории.

Синтаксис

```
file.openNextFile()
```

Параметры

file: экземпляр класса File, ссылающийся на директорию.

Возвращаемые значения

char: имя следующего файла или папки в пределах директории.

Пример

```
#include <SD.h>

File root;

void setup()
{
  Serial.begin(9600);
  pinMode(10, OUTPUT);

  SD.begin(10);

  root = SD.open("/");

  printDirectory(root, 0);

  Serial.println("done!");
}

void loop()
{
  // после первоначальной настройки ничего не происходит.
}

void printDirectory(File dir, int numTabs) {
```

```
  while(true) {

    File entry =  dir.openNextFile();
    if (! entry) {
      // больше нет файлов
      Serial.println("**nomorefiles**");
    }
    for (uint8_t i=0; i<numTabs; i++) {
      Serial.print('\t');
    }
    Serial.print(entry.name());
    if (entry.isDirectory()) {
      Serial.println("/");
      printDirectory(entry, numTabs+1);
    } else {
      // у файлов есть размер, в отличие от директорий
      Serial.print("\t\t");
      Serial.println(entry.size(), DEC);
    }
  }
}
```

Смотрите также

isDirectory()
rewindDirectory()

rewindDirectory()

Описание

Функция rewindDirectory() позволяет вернуться к первому файлу в пределах директории. Как правило, используется вместе с функцией openNextFile().

Синтаксис

```
file.rewindDirectory()
```

Параметры

file: экземпляр класса File.

Возвращаемые значения

нет

Пример

```
#include <SD.h>

File root;

void setup()
{
  Serial.begin(9600);
  pinMode(10, OUTPUT);

  SD.begin(10);

  root = SD.open("/");

  printDirectory(root, 0);

  Serial.println("done!");
}

void loop()
{
  // после первоначальной настройки ничего не происходит.
}
```

```
void printDirectory(File dir, int numTabs) {
   while(true) {

     File entry =  dir.openNextFile();
     if (! entry) {
       // больше нет файлов
       Serial.println("**nomorefiles**");
     }
     for (uint8_t i=0; i<numTabs; i++) {
       Serial.print('\t');
     }
     Serial.print(entry.name());
     if (entry.isDirectory()) {
       Serial.println("/");
       printDirectory(entry, numTabs+1);
     } else {
       // у файлов есть размер, в отличие от директорий
       Serial.print("\t\t");
       Serial.println(entry.size(), DEC);
     }
   }
}
```

Смотрите также

[open()](#)
[openNextFile()](#)
[isDirectory()](#)

Библиотека SPI

Данная библиотека позволяет Ардуино взаимодействовать с различными SPI-устройствами, выступая при этом в роли ведущего устройства.

Краткое введение в интерфейс SPI (Serial Peripheral Interface)

Последовательный периферийный интерфейс (SPI) - это синхронный протокол последовательной передачи данных, используемый для связи микроконтроллера с одним или несколькими периферийными устройствами. Интерфейс SPI отличается относительно высокой скоростью и предназначен для связи близко расположенных устройств. Он также может использоваться для взаимодействия двух микроконтроллеров.

Согласно протоколу SPI, одно из взаимодействующих устройств (обычно микроконтроллер) всегда является ведущим и контролирует ведомые периферийные устройства. Как правило, все взаимодействующие устройства объединены тремя общими линиями:

MISO (Master In Slave Out) - линия для передачи данных от ведомого устройства (Slave) к ведущему (Master),
MOSI (Master Out Slave In) - линия для передачи данных от ведущего устройства (Master) к ведомым (Slave),
SCK (Serial Clock) - тактовые импульсы, генерируемые ведущим устройством (Master) для синхронизации процесса передачи данных.

Помимо перечисленных, на каждое устройство отводится отдельная линия:

SS (Slave Select) - вывод, присутствующий на каждом ведомом устройстве. Он предназначен для активизации Мастером того или иного периферийного устройства.

Периферийное устройство (Slave) взаимодействует с ведущим (Master) тогда, когда на выводе SS присутствует низкий уровень сигнала. В противном случае данные от Master-устройства будут игнорироваться. Такая архитектура позволяет взаимодействовать с несколькими SPI-устройствами, подключенными к одной и той же шине: MISO, MOSI и SCK.

Перед тем, как отправлять данные новому SPI-устройству, необходимо выяснить о нем несколько основных моментов:

Сдвиг данных должен осуществляться, начиная со старшего бита (MSB) или с младшего бита (LSB)? Порядок следования данных контролируется функцией **SPI.setBitOrder()**.
При отсутствии тактовых импульсов линия SCK должна находиться в высоком или низком уровне? Считывание данных происходит по фронту или по спаду тактового импульса? Эти режимы работы контролируются функцией **SPI.setDataMode()**.
Какова должна быть скорость передачи данных по SPI? Этот параметр контролируется функцией **SPI.setClockDivider()**.

Поскольку стандарт SPI является открытым, его реализация в разных устройствах может немного отличаться. Поэтому при написании программ, особое внимание необходимо уделять даташиту того или иного устройства.

Грубо говоря, существует четыре режима передачи данных, отличающиеся условием сдвига данных (по фронту или по спаду синхро-импульсов - так называемая **фаза**), а также уровнем сигнала, в котором должна находится линия SCK при отсутствии синхро-импульсов (**полярность**). Различные комбинации фазы и полярности, формирующие четыре режима передачи данных, сведены в таблицу:

Режим	Полярность (CPOL)	Фаза (CPHA)
SPI_MODE0	0	0
SPI_MODE1	0	1
SPI_MODE2	1	0
SPI_MODE3	1	1

Для изменения режима передачи данных служит функция SPI.setDataMode().

Каждое SPI-устройство налагает определенные ограничения на максимальную скорость SPI-шины. Для корректной работы периферийных устройств в библиотеке предусмотрена функция SPI.setClockDivider(), позволяющая изменять тактовую частоту шины (по умолчанию 4 МГц).

После правильной настройки всех параметров SPI, останется только выяснить, какие регистры периферийного устройства отвечают за те или иные его функции. Как правило, это описано в даташите устройства.

Для получения дополнительной информации об интерфейсе SPI, см. страницу Википедии.

Соединения

Ниже в таблице приведены номера выводов, использующиеся шиной SPI в тех или иных моделях Ардуино:

Плата Arduino	**MOSI**	**MISO**	**SCK**	**SS (slave)**	**SS (master)**
Uno или Duemilanove	11 или ICSP-4	12 или ICSP-1	13 или ICSP-3	10	-
Mega1280 или Mega2560	51 или ICSP-4	50 или ICSP-1	52 или ICSP-3	53	-
Leonardo	ICSP-4	ICSP-1	ICSP-3	-	-
Due	ICSP-4	ICSP-1	ICSP-3	-	4, 10, 52

Обратите внимание, что на всех платах выводы MISO, MOSI и SCK соединены с одними и теми же контактами разъема ICSP. Такое расположение может быть удобно при создании универсальных плат расширения, работающих на всех моделях Ардуино.

Особенности работы вывода SS в Ардуино на базе AVR

У всех моделей Ардуино на основе микроконтроллеров AVR есть вывод SS, который используется в режиме работы **Slave** (например, при управлении Ардуино внешним ведущим устройством). Однако, в библиотеке реализован только режим работы Master, поэтому в этом режиме вывод SS должен быть сконфигурирован как выход. В противном случае SPI может аппаратно переключиться в режим Slave, что приведет к неработоспособности функций библиотеки.

Для управления выводом SS периферийных устройств можно использовать любой из доступных выводов. Например, на плате расширения Arduino Ethernet для взаимодействия со встроенной SD-картой и контроллером Ethernet по SPI используются выводы 4 и 10 соответственно.

Расширенные возможности SPI на Arduino Due

Существуют некоторые особенности работы с интерфейсом SPI на платах Arduino Due. Помимо основных функций и методов, применимых ко всем платам Ардуино, в библиотеке SPI предусмотрено несколько **дополнительных методов**. Эти методы реализовывают аппаратные возможности микроконтроллеров SAM3X и предоставляют разработчику расширенные возможности:

автоматический управление процессом выбора ведомого устройства;
автоматическое управление конфигурациями интерфейса SPI для различных устройств (тактовая частота, режим передачи данных и т.д.). Благодаря этому каждое из ведомых устройств может иметь собственный набор настроек, автоматически применяемых в начале передачи.

В Arduino Due есть три отдельных вывода (4, 10 и 52) для управления линиями SS периферийных устройств.

Использование расширенных возможностей SPI на Arduino Due.

begin()

Описание

Функция инициализирует работу шины SPI, а именно: конфигурирует выводы SCK, MOSI и SS как выходы, формирует низкий уровень сигнала на выводах SCK и MOSI, и высокий уровень - на выводе SS.

Дополнительные возможности Arduino Due

На Arduino Due в качестве параметра метода begin() можно указать один из аппаратных выводов SS. В этом случае работой этого вывода будет управлять непосредственно контроллер SPI-интерфейса.

Следует иметь ввиду, что в этом случае указанный вывод нельзя использовать в качестве вывода общего назначения до тех пор, пока не будет вызван метод SPI.end().

На Arduino Due выводами SS, работой которых может аппаратно управлять контроллер SPI-интерфейса, являются выводы: 4, 10, 52 и 54 (A0).

Синтаксис

```
SPI.begin()
SPI.begin(slaveSelectPin)          (только для Arduino Due)
```

Параметры

slaveSelectPin: вывод SS ведомого устройства (slave) - *(только для Arduino Due)*

Возвращаемые значения

нет

Смотрите также

SPI.end()
SPI.setClockDivider()
SPI.setDataMode()
SPI.setBitOrder()

Использование расширенных возможностей SPI на Arduino Due

end()

Описание

Функция завершает работу шины SPI (режим работы выводов SPI остается прежним).

Дополнительные возможности Arduino Due

В качестве параметра метода end() можно указать один из аппаратных выводов SS Arduino Due. В этом случае указанный вывод отсоединяется от интерфейса SPI и вновь становится выводом общего назначения.

Синтаксис

```
SPI.end()
SPI.end(slaveSelectPin)         (только для Arduino Due)
```

Параметры

slaveSelectPin: вывод SS ведомого устройства (slave) - *(только для Arduino Due)*

Возвращаемые значения

нет

Смотрите также

begin()

setBitOrder()

Описание

Функция определяет порядок следования бит данных по шине SPI: младшим битом вперед (LSBFIRST) или старшим битом вперед (MSBFIRST).

Дополнительные возможности Arduino Due

В качестве параметра метода setBitOrder() можно указать один из аппаратных выводов SS Arduino Due. В этом случае порядок следования бит данных задается только для того устройства, которое соединено с указанным выводом. Подробнее о расширенных возможностях SPI на Arduino Due см. здесь.

Синтаксис

```
SPI.setBitOrder(order)
SPI.setBitOrder(slaveSelectPin, order)        (только для Arduino Due)
```

Параметры

order: одно из двух значений - LSBFIRST или MSBFIRST.

Возвращаемые значения

нет

Смотрите также

setClockDivider()
setDataMode()

setClockDivider()

Описание

Позволяет задать тактовую частоту SPI, указав коэффициент деления тактовой частоты контроллера. В Ардуино на базе AVR-микроконтроллеров можно использовать один из следующих коэффициентов деления: 2, 4, 8, 16, 32, 64 или 128. По умолчанию тактовая частота SPI в четыре раза меньше тактовой частоты контроллера (SPI_CLOCK_DIV4). Т.е., если тактовая частота контроллера 16 МГц, то SPI будет работать на частоте 4 МГц.

Arduino Due
В Arduino Due системную частоту можно делить на любое число в диапазоне от 1 до 255. По умолчанию установлен коэффициент 21, чтобы частота SPI была равной 4 МГц, как и на других моделях Ардуино.

Расширенные возможности в Arduino Due

Если при вызове функции setClockDivider() вы укажите один из выводов SS Arduino Due, то указанная вами частота будет задана только для того устройства на шине SPI, которое соединено с эти выводом. Об этой и других возможностях Arduino Due подробнее см. здесь.

Синтаксис

```
SPI.setClockDivider(divider)
SPI.setClockDivider(slaveSelectPin, divider)      (только для Arduino Due)
```

Параметры

divider:	SPI_CLOCK_DIV2 SPI_CLOCK_DIV4 SPI_CLOCK_DIV8 SPI_CLOCK_DIV16 SPI_CLOCK_DIV32 SPI_CLOCK_DIV64 SPI_CLOCK_DIV128	(*Для AVR-устройств*)

slaveSelectPin:	вывод SS	(Только для *Arduino Due*)
divider:	число от 1 до 255	(Только для *Arduino Due*)

Возвращаемые значения

нет

Смотрите также

setDataMode()
setBitOrder()

setDataMode()

Описание

Позволяет задать режим работы SPI: полярность и фазу тактовых импульсов. Для получения подробной информации см. статью на Wikipedia.

Расширенные возможности в Arduino Due

Если при вызове функции setDataMode() вы укажите один из выводов SS Arduino Due, то указанный вами режим будет задан только для того устройства на шине SPI, которое соединено с эти выводом. Об этой и других возможностях Arduino Due подробнее см. здесь.

Синтаксис

```
SPI.setDataMode(mode)
SPI.setDataMode(slaveSelectPin, mode)    (Только для Arduino Due)
```

Параметры

mode:	SPI_MODE0 SPI_MODE1 SPI_MODE2 SPI_MODE3	
slaveSelectPin	slave device SS pin	(Только для *Arduino Due*)

Возвращаемые значения

нет

Смотрите также

setClockDivider()
setBitOrder()

transfer()

Описание

Осуществляет передачу байта данных по шине SPI, одновременно принимая входящий байт.

Расширенные возможности в Arduino Due

Если при вызове функции SPI.transfer() вы укажите один из выводов SS Arduino Due, то указанный вами вывод будет автоматически активирован (установлен в низкий уровень) перед началом передачи и обратно деактивирован (установлен в высокий уровень) после завершения передачи данных.

Для управления состоянием вывода SS после передачи данных предусмотрены специальные параметры - SPI_CONTINUE и SPI_LAST. При использовании SPI_CONTINUE, вывод SS будет оставаться активным (в низком уровне) даже после передачи, что позволяет продолжить отправку байтов данных функцией transfer() в пределах той же самой транзакции. При отправке последнего байта из транзакции необходимо использовать параметр SPI_LAST. Если третий параметр в функции transfer() не указан, то в качестве этого параметра по умолчанию используется SPI_LAST. После завершения передачи с флагом SPI_LAST, вывод SS становится неактивным (снова переходит в высокий уровень).

Подробнее о расширенных возможностях Arduino Due см. здесь.

Синтаксис

```
SPI.transfer(val)
SPI.transfer(slaveSelectPin, val)     (только для Arduino Due)
SPI.transfer(slaveSelectPin, val, transferMode)  (только для
Arduino Due)
```

Параметры

val:	байт данных, который необходимо отправить по SPI	
slaveSelectPin:	вывод SS	(только *Arduino Due*)
transferMode:	SPI_CONTINUE: оставляет вывод SS в низком уровне, что позволяет продолжить передачу байтов.	

SPI_LAST: *значение по умолчанию* - после передачи одного байта данных, вывод SS возвращается в высокий уровень.

Возвращаемые значения

байт данных, полученный по шине SPI

Использование расширенных возможностей SPI на Arduino Due

Микроконтроллер SAM3X на Arduino Due предоставляет разработчику расширенные возможности по работе с интерфейсом SPI. При этом можно использовать как расширенное API, так и традиционный подходы, характерные платам на базе AVR-микроконтроллеров.

Расширенное API в качестве линий CS позволяет использовать выводы 4, 10 и 52.

Как пользоваться

Вначале необходимо указать выводы, которые будут использоваться в качестве линий CS для каждого SPI-устройства.

Arduino Due может автоматически управлять этими выводами, распределяя доступ к шине SPI между ведомыми устройствами. При этом каждое устройство может иметь индивидуальные настройки SPI-интерфейса, такие, как режим работы и скорость передачи данных.

Таким образом, для управления несколькими ведомыми устройствами необходимо объявить соответствующие им выводы CS в функции setup(). Ниже показан пример работы с двумя устройствами, расположенными на одной SPI-шине. Вывод CS одного устройства подключен к выводу 4, другого - к выводу 10.

```
void setup(){
  // инициализируем шину для устройства, подключенного к выводу
4
  SPI.begin(4);
  // инициализируем шину для устройства, подключенного к выводу
10
  SPI.begin(10);
}
```

После объявления выводов CS, каждому ведомому устройству можно задать индивидуальные настройки интерфейса SPI. Например, если устройства работают на разной тактовой частоте, функция setup() будет выглядеть следующим образом:

```
void setup(){
  // инициализируем шину для устройства, подключенного к выводу
4
  SPI.begin(4);
  // устанавливаем для этого устройства коэффициент деления
тактовой частоты 21
```

```
  SPI.setClockDivider(4, 21);
  // инициализируем шину для устройства, подключенного к выводу
10
  SPI.begin(10);
  // устанавливаем для этого устройства коэффициент деления
тактовой частоты 84
  SPI.setClockDivider(10, 84);
}
```

Простая передача байта ведомому устройству, подключенному к выводу 4, будет выглядеть так:

```
void loop(){
  byte response = SPI.transfer(4, 0xFF);
}
```

В результате выполнения этого кода, значение "0xFF" будет отправлено SPI-устройству, подключенному к выводу 4, а ответные данные, пришедшие от устройства по линии MISO, будут помещены в переменную *response*. Управление линией CS осуществляется контроллером интерфейса SPI автоматически. Таким образом, команда transfer выполняет следующие операции:

активизирует ведомое устройство, формируя на выводе 4 низкий уровень сигнала (LOW)
отправляет значение 0xFF по шине SPI и возвращает полученный байт
отключает ведомое устройство от шины SPI, формируя на выводе 4 высокий уровень сигнала (HIGH)

Также возможна отправка нескольких байт за одну транзакцию. Для этого команде transfer необходимо указать параметр, который заставит ее не отключать ведомое устройство после передачи байта:

```
void loop(){
//передаем 0x0F устройству, подключенному к выводу 10, оставляя
его активным
SPI.transfer(10, 0xF0, SPI_CONTINUE);

//передаем 0x00 устройству, подключенному к выводу 10, оставляя
его активным
SPI.transfer(10, 0x00, SPI_CONTINUE);

//передаем 0x00 устройству, подключенному к выводу 10, и
сохраняем полученный
//байт в переменной response1. По прежнему оставляем устройство
активным
byte response1 = SPI.transfer(10, 0x00, SPI_CONTINUE);

//передаем 0x00 устройству, подключенному к выводу 10, и
```

```
сохраняем полученный
//байт в переменной response2, отключая устройство.
byte response2 = SPI.transfer(10, 0x00);
}
```

Параметр SPI_CONTINUE заставляет ведомое устройство быть активным между передачами. При передаче последнего байта этот параметр не указывается.

См. справку по функциям setClockDivider(), setDataMode(), transfer(), setBitOrder() для получения дополнительной информации об их синтаксисе при использовании расширенного API.

Примечание: после вызова функции SPI.begin(), вывод, переданный ей в качестве параметра, нельзя использовать в качестве вывода общего назначения.

Библиотека SoftwareSerial

В Ардуино реализована аппаратная поддержка интерфейса последовательной передачи данных через выводы 0 и 1 (которые также используются для связи с компьютером посредством USB). Аппаратная работа с последовательным интерфейсом осуществляется с помощью встроенного в микроконтроллер специального устройства, называемого приемопередатчиком UART. Он позволяет микроконтроллеру Atmega обрабатывать поступающие данные даже во время работы над другими задачами.

Библиотека SoftwareSerial позволяет реализовать последовательный интерфейс на любых цифровых выводах Ардуино с помощью программных средств, дублирующих функциональность UART (отсюда и название "SoftwareSerial"). Библиотека позволяет программно создавать несколько последовательных портов, работающих на скорости до 115200 бод. Для устройств, работающих с инвертированным сигналом, в библиотеке предусмотрен соответствующий параметр, включающий инвертирование.

Начиная с версии 1.0, SoftwareSerial основывается на библиотеке NewSoftSerial автора Mikal Hart.

Ограничения

Среди известных ограничений библиотеки SoftwareSerial можно перечислить следующие:

При использовании нескольких последовательных портов, в каждый момент времени только один из них может получать данные.
На платах Arduino Mega и Mega2560 некоторые выводы не поддерживают прерывания, возникающие при изменении уровня сигнала. В силу этого, на данных платах в качестве вывода RX могут использоваться только следующие выводы: 10, 11, 12, 13, 14, 15, 50, 51, 52, 53, A8 (62), A9 (63), A10 (64), A11 (65), A12 (66), A13 (67), A14 (68), A15 (69).
На Arduino Leonardo некоторые выводы не поддерживают прерывания, возникающие при изменении уровня сигнала. Поэтому, на этой плате в качестве вывода RX могут использоваться только следующие выводы: 8, 9, 10, 11, 14 (MISO), 15 (SCK), 16 (MOSI).

Пример

```
/*
  Программа тестирования последовательных портов, создаваемых с
помощью
  библиотеки SoftwareSerial

 Данные, получаемые аппаратным портом, отправляются на
```

```
программный порт.
 Данные, получаемые программным портом, отправляются на
аппаратный порт.

 Схема:
 * RX - цифровой вывод 10 (необходимо соединить с выводом TX
другого устройства)
 * TX - цифровой вывод 11 (необходимо соединить с выводом RX
другого устройства)

 Примечания:
 На платах Arduino Mega и Mega2560 некоторые выводы не
поддерживают прерывания,
 возникающие при изменении уровня сигнала. Поэтому, на данных
платах в качестве
 вывода RX могут использоваться только следующие выводы: 10, 11,
12, 13, 14, 15, 50,
 51, 52, 53, A8 (62), A9 (63), A10 (64), A11 (65), A12 (66), A13
(67), A14 (68), A15 (69).



 На Arduino Leonardo некоторые выводы не поддерживают
прерывания, возникающие при
 изменении уровня сигнала. Поэтому, на этой плате в качестве
вывода RX могут
 использоваться только следующие выводы: 8, 9, 10, 11, 14
(MISO), 15 (SCK), 16 (MOSI).

 дата создания не известна
 модифицировано 25 мая 2012
 Автор: Tom Igoe
 на основе примера Mikal Hart

 Данный код открыт для использования.

 */
#include <SoftwareSerial.h>

SoftwareSerial mySerial(10, 11); // RX, TX

void setup()
{
  // Инициализируем последовательный интерфейс и ждем открытия
порта:
  Serial.begin(57600);
  while (!Serial) {
    ; // ожидаем подключения к последовательному порту.
Необходимо только для Leonardo
```

```
  }


  Serial.println("Goodnight moon!");

  // устанавливаем скорость передачи данных для
последовательного порта, созданного
  // библиотекой SoftwareSerial
  mySerial.begin(4800);
  mySerial.println("Hello, world?");
}

void loop() // выполняется циклически
{
  if (mySerial.available())
    Serial.write(mySerial.read());
  if (Serial.available())
    mySerial.write(Serial.read());
}
```

SoftwareSerial(rxPin, txPin)

Описание

Функция SoftwareSerial(rxPin, txPin) создает новый объект типа SoftwareSerial и присваивает его указанной переменной (см. пример ниже).

Для начала связи служит функция SoftwareSerial.begin().

Параметры

rxPin: вывод для приема данных по последовательному интерфейсу

txPin: вывод для передачи данных по последовательному интерфейсу

Пример

```
#define rxPin 2
#define txPin 3

// инициализируем новый последовательный порт
SoftwareSerial mySerial =  SoftwareSerial(rxPin, txPin);
```

Смотрите также

SoftwareSerial.begin()
SoftwareSerial.read()
SoftwareSerial.print()
SoftwareSerial.println()

SoftwareSerial: available()

Описание

Возвращает количество непрочитанных байт (символов), принятых через программный последовательный порт. Непрочитанные данные накапливаются во входном последовательном буфере.

Синтаксис

```
mySerial.available()
```

Параметры

нет

Возвращаемые значения

количество непрочитанных байт

Пример

```
// подключаем библиотеку SoftwareSerial для использования ее функций:
#include <SoftwareSerial.h>

#define rxPin 10
#define txPin 11

// инициализируем новый последовательный порт
SoftwareSerial mySerial =  SoftwareSerial(rxPin, txPin);

void setup()  {
  // задаем режим работы выводов tx, rx:
  pinMode(rxPin, INPUT);
  pinMode(txPin, OUTPUT);
  // устанавливаем скорость передачи данных последовательного порта
  mySerial.begin(9600);
}

void loop() {
  if (mySerial.available()>0){
  mySerial.read();
```

```
  }
}
```

Смотрите также

SoftwareSerial()
read()
print()
println()

SoftwareSerial: begin(speed)

Описание

Задает скорость передачи данных (в бодах) последовательного порта. Поддерживаемые значения: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 31250, 38400, 57600 и 115200.

Параметры

speed: скорость передачи данных в бодах (*long*)

Возвращаемые значения

нет

Пример

```
// подключаем библиотеку SoftwareSerial для использования ее
функций:
#include <SoftwareSerial.h>

#define rxPin 10
#define txPin 11

// инициализируем новый последовательный порт
SoftwareSerial mySerial =  SoftwareSerial(rxPin, txPin);

void setup()  {
  // задаем режим работы выводов tx, rx:
  pinMode(rxPin, INPUT);
  pinMode(txPin, OUTPUT);
  // устанавливаем скорость передачи данных последовательного
порта
  mySerial.begin(9600);
}

void loop() {
  // ...
 }
}
```

Смотрите также

SoftwareSerial()
read()
print()
println()

SoftwareSerial: isListening()

Описание

Позволяет проверить, активен ли программный последовательный порт. Функция возвращает true. если порт находится в режиме ожидания данных.

Синтаксис

```
mySerial.isListening()
```

Параметры

нет

Возвращаемые значения

boolean

Пример

```
#include <SoftwareSerial.h>

// программный последовательный порт : TX = цифровой вывод 10,
RX = цифровой вывод 11
SoftwareSerial portOne(10,11);

void setup()
{
  // инициализируем аппаратный последовательный порт
  Serial.begin(9600);

  // инициализируем программный последовательный порт
  portOne.begin(9600);
}

void loop()
{
  if (portOne.isListening()) {
   Serial.println("Port One is listening!");
}
```

Смотрите также

SoftwareSerial()
read()
print()
println()

SoftwareSerial: overflow()

Описание

Проверяет входной буфер программного последовательного порта на предмет его переполнения. При вызове этой функции, флаг переполнения буфера сбрасывается. Поэтому при всех последующих вызовах, функция будет возвращать false до тех пор, пока не будет принят (и проигнорирован) очередной байт данных.

Входной буфер последовательного порта рассчитан на 64 байта.

Синтаксис

```
mySerial.overflow()
```

Параметры

нет

Возвращаемые значения

boolean

Пример

```
#include <SoftwareSerial.h>

// программный последовательный порт : TX = цифровой вывод 10,
RX = цифровой вывод 11
SoftwareSerial portOne(10,11);

void setup()
{
  // инициализируем аппаратный последовательный порт
  Serial.begin(9600);

  // инициализируем программный последовательный порт
  portOne.begin(9600);
}

void loop()
{
  if (portOne.overflow()) {
```

```
   Serial.println("SoftwareSerial overflow!");
}
```

Смотрите также

SoftwareSerial()
read()
print()
println()

SoftwareSerial: peek

Описание

Возвращает символ, принятый программным последовательным портом через вывод RX. Однако, в отличие от read(), многократный вызов данной функции будет приводить к получению одного и того же значения.

Следует иметь ввиду, что в каждый момент времени принимать поступающие данные может только один программный порт. Для выбора принимающего порта (экземпляра класса SoftwareSerial) используйте функцию listen().

Параметры

нет

Возвращаемые значения

полученный символ, или -1, если такового нет

Пример

```
SoftwareSerial mySerial(10,11);

void setup()
{
  mySerial.begin(9600);
}

void loop()
{
  char c = mySerial.peek();
}
```

Смотрите также

SoftwareSerial()
begin()
print()
println()
read()

SoftwareSerial: read

Описание

Возвращает символ, принятый программным последовательным портом через вывод RX. Следует иметь ввиду, что в каждый момент времени принимать поступающие данные может только один программный порт. Для выбора принимающего порта (экземпляра класса SoftwareSerial) используйте функцию listen().

Параметры

нет

Возвращаемые значения

полученный символ, или -1, если такового нет

Пример

```
SoftwareSerial mySerial(10,11);

void setup()
{
  mySerial.begin(9600);
}

void loop()
{
  char c = mySerial.read();
}
```

Смотрите также

SoftwareSerial()
begin()
print()
println()

SoftwareSerial: print(data)

Описание

Выводит данные через вывод TX программного последовательного порта. Работа данной функции аналогична функции Serial.print().

Параметры

Могут варьироваться, см. описание функции Serial.print().

Возвращаемые значения

byte

Функция print() возвращает количество отправленных байт. Считывание этого значения не обязательно.

Пример

```
SoftwareSerial serial(10,11);
int analogValue;

void setup()
{
  serial.begin(9600);
}

void loop()
{
  // считываем значение с аналогового входа 0:
  analogValue = analogRead(A0);

  // выводим его в разных форматах:
  serial.print(analogValue);          // выводим как ASCII-
символы в десятичном виде
  serial.print("\t");                 // выводим символ табуляции
  serial.print(analogValue, DEC);     // выводим как ASCII-
символы в десятичном виде
  serial.print("\t");                 // выводим символ табуляции
  serial.print(analogValue, HEX);     // выводим как ASCII-
символы в шестнадцатеричном виде
  serial.print("\t");                 // выводим символ табуляции
  serial.print(analogValue, OCT);     // выводим как ASCII-
символы в восьмеричном виде
```

```
  serial.print("\t");                 // выводим символ табуляции
  serial.print(analogValue, BIN);     // выводим как ASCII-
символы в двоичном виде
  serial.print("\t");                 // выводим символ табуляции
  serial.print(analogValue/4, BYTE); // выводим в необработанном
виде (предварительно
                                      // поделив на 4, т.к.
analogRead() возвращает значения
                                      // в диапазоне от 0 до
1023, а в байте данных может
                                      // хранится число не больше
255)
  serial.print("\t");                 // выводим символ
табуляции
  serial.println();                   // выводим символ перевода
строки

  // задержка 10 миллисекунд перед очередным считыванием:
  delay(10);
}
```

Смотрите также

SoftwareSerial()
begin()
read()
println()

SoftwareSerial: println(data)

Описание

Выводит данные через вывод TX программного последовательного порта с последующим символом возврата каретки и перевода строки. Работа данной функции аналогична функции Serial.println().

Параметры

Могут варьироваться, см. описание функции Serial.println().

Возвращаемые значения

byte

Функция println() возвращает количество отправленных байт. Считывание этого значения не обязательно.

Пример

```
SoftwareSerial serial(10,11);
int analogValue;

void setup()
{
  serial.begin(9600);
}

void loop()
{
  // считываем значение с аналогового входа 0:
  analogValue = analogRead(A0);

  // выводим его в разных форматах:
  serial.print(analogValue);          // выводим как ASCII-
символы в десятичном виде
  serial.print("\t");                 // выводим символ табуляции
  serial.print(analogValue, DEC);     // выводим как ASCII-
символы в десятичном виде
  serial.print("\t");                 // выводим символ табуляции
  serial.print(analogValue, HEX);     // выводим как ASCII-
символы в шестнадцатеричном виде
  serial.print("\t");                 // выводим символ табуляции
  serial.print(analogValue, OCT);     // выводим как ASCII-
```

```
символы в восьмеричном виде
  serial.print("\t");                 // выводим символ табуляции
  serial.print(analogValue, BIN);     // выводим как ASCII-
символы в двоичном виде
  serial.print("\t");                 // выводим символ табуляции
  serial.print(analogValue/4, BYTE); // выводим в необработанном
виде (предварительно
                                      // поделив на 4, т.к.
analogRead() возвращает значения
                                      // в диапазоне от 0 до
1023, а в байте данных может
                                      // хранится число не больше
255)
  serial.print("\t");                 // выводим символ
табуляции
  serial.println();                   // выводим символ перевода
строки

  // задержка 10 миллисекунд перед очередным считыванием:
  delay(10);
}
```

Смотрите также

SoftwareSerial()
begin()
read()
print()

SoftwareSerial: listen()

Описание

Переводит указанный последовательный порт в режим ожидания данных. В каждый момент времени только один программный порт может принимать данные; при этом данные, поступающие другим портам, будут игнорироваться. Если при вызове функции listen() текущий активный порт изменяется на другой, то все принятые ранее данные стираются.

Синтаксис

```
mySerial.listen()
```

Параметры

mySerial: имя экземпляра класса SoftwareSerial, который должен принимать данные

Возвращаемые значения

нет

Пример

```
#include <SoftwareSerial.h>

// программный последовательный порт : TX = цифровой вывод 10,
RX = цифровой вывод 11
SoftwareSerial portOne(10, 11);

// программный последовательный порт : TX = цифровой вывод 8, RX
= цифровой вывод 9
SoftwareSerial portTwo(8, 9);

void setup()
{
  // инициализируем аппаратный последовательный порт
  Serial.begin(9600);

  // инициализируем оба программных порта
  portOne.begin(9600);
  portTwo.begin(9600);
```

```
}

void loop()
{
  portOne.listen();

  if (portOne.isListening()) {
   Serial.println("Port One is listening!");
}else{
   Serial.println("Port One is not listening!");
}

  if (portTwo.isListening()) {
   Serial.println("Port Two is listening!");
}else{
   Serial.println("Port Two is not listening!");
}

}
```

Смотрите также

SoftwareSerial()
read()
print()
println()

SoftwareSerial: write(data)

Описание

Выводит данные в виде последовательности байт через вывод TX программного последовательного порта. Работа данной функции аналогична функции Serial.write().

Параметры

См. описание функции Serial.write().

Возвращаемые значения

byte

Функция write() возвращает количество записанных байт. Считывание этого значения не обязательно.

Пример

```
SoftwareSerial mySerial(10, 11);

void setup()
{
  mySerial.begin(9600);
}

void loop()
{
  mySerial.write(45); // отправляем байт данных со значением 45

   int bytesSent = mySerial.write("hello"); //отправляем строку "hello", считывая длину этой строки.
}
```

Смотрите также

SoftwareSerial()
begin()
read()
print()

Библиотека Wire

Данная библиотека позволяет Ардуино взаимодействовать с различными устройствами по интерфейсу I2C / TWI. На платах Ардуино версии R3 (с распиновкой 1.0) линии SDA (данные) и SCL (тактовые импульсы), связанные с этим интерфейсом, расположены на разъеме возле контакта AREF. В Arduino Due реализовано два интерфейса I2C / TWI, линии одного из них (SDA1 и SCL1) расположены возле вывода AREF, линии второго - на выводах 20 и 21.

Расположение выводов TWI на тех или иных платах Ардуино для наглядности сведено в таблицу:

Ардуино	**Выводы I2C / TWI**
Uno, Ethernet	A4 (SDA), A5 (SCL)
Mega2560	20 (SDA), 21 (SCL)
Leonardo	2 (SDA), 3 (SCL)
Due	20 (SDA), 21 (SCL), SDA1, SCL1

Начиная с версии языка Arduino 1.0, библиотека Wire наследует функции класса Stream, что позволяет ей быть совместимой с другими библиотеками, осуществляющими запись и чтение данных. Поэтому, методы send() и receive() были заменены методами read() и write().

Примечание

Согласно протоколу I2C, адрес устройства может состоять как из 7, так и из 8 бит. Как правило, 7 бит идентифицируют устройство, в то время, как восьмой бит задает направление передачи данных: от устройства (чтение) или к нему (запись). Все функции библиотеки Wire используют 7-битную адресацию. Поэтому, при работе с устройством, использующим 8-битную адресацию, вам придется отбрасывать младший бит (например, сдвигая значение на один бит вправо), тем самым ограничивая диапазон возможных адресов в пределах 0 - 127.

Wire.begin()

Wire.begin(address)

Описание

Инициализирует библиотеку Wire и подключает Ардуино к шине I2C в роли ведущего (master) или ведомого (slave) устройства. Как правило, эта функция вызывается только один раз.

Параметры

address: 7-битный адрес ведомого устройства (не обязательный параметр); если адрес не указан, то Ардуино выступает в роли ведущего устройства (master).

Возвращаемые значения

нет

Wire.requestFrom()

Описание

Функция запрашивает данные у ведомого устройства (slave); как правило, используется только ведущим устройством (Master). После вызова requestFrom () запрашиваемые данные должны быть считаны с помощью функций available () и read().

Начиная с версии Ардуино 1.0.1, функция requestFrom() может принимать третий параметр - логическое значение, обеспечивающее лучшую совместимость с некоторыми I2C-устройствами.

Если этот параметр равен true, то функция requestFrom() отправит запрос со стоповым битом, что позволит освободить шину I2C.

Если этот параметр равен false, то после отправки запроса шина по прежнему будет занята, что предотвратит отправку посторонних сообщений другими ведущими устройствами. Этот режим позволяет Мастеру отправлять по несколько запросов за один сеанс.

Значение по умолчанию - true.

Синтаксис

```
Wire.requestFrom(address, quantity)
Wire.requestFrom(address, quantity, stop)
```

Параметры

address: 7-битный адрес ведомого устройства, у которого запрашиваются данные

quantity: количество запрашиваемых байт

stop: boolean. При значении true будет отправлен запрос со стоповым битом, что позволит освободить шину. При значении false - соединение будет поддерживаться в активном состоянии.

Возвращаемые значения

byte : количество байт, возвращенных ведомым устройством

Смотрите также

Wire.available()

Wire.read()

Wire.beginTransmission(address)

Описание

Начинает процедуру передачи данных по интерфейсу I2C ведомому устройству с указанным адресом. Для последующей отправки данных, необходимо сперва поставить их в очередь с помощью функции write(), после чего осуществить, непосредственно, передачу функцией endTransmission().

Параметры

address: 7-битный адрес принимающего устройства

Возвращаемые значения

нет

Смотрите также

Wire.write()
Wire.endTransmission()

Wire.endTransmission()

Описание

Завершает процедуру передачи данных ведомому устройству, инициированную функцией beginTransmittion(). При этом функция отправляет байты, поставленные в очередь функцией write().

Начиная с версии Ардуино 1.0.1, функция endTransmission() может принимать логический параметр, способствующий лучшей совместимости с некоторыми I2C-устройствами.

Если этот параметр равен true, то функция requestFrom() отправит запрос со стоповым битом, что позволит освободить шину I2C.

Если этот параметр равен false, то после отправки запроса шина по-прежнему будет занята, что предотвратит отправку посторонних сообщений другими ведущими устройствами. Этот режим позволяет Мастеру отправлять по несколько запросов за один сеанс.

Значение по умолчанию - true.

Синтаксис

```
Wire.endTransmission()
Wire.endTransmission(stop)
```

Параметры

stop: boolean. При значении true будет отправлен запрос со стоповым битом, что позволит освободить шину. При значении false - соединение будет поддерживаться в активном состоянии.

Возвращаемые значения

byte, байт данных, характеризующий статус передачи:

0: передача успешна
1: объем данных слишком велик для буфера передачи
2: получен NACK при передаче адреса
3: получен NACK при передаче данных
4: другая ошибка

Смотрите также

Wire.beginTransmission()

Wire.write()

write()

Описание

На ведомом устройстве (Slave) данная функция отправляет данные в ответ на запрос ведущего устройства. На ведущем устройстве (Master) функция добавляет данные в очередь отправки для последующей передачи ведомому устройству (в этом случае функция write() должна вызываться между beginTransmission() и endTransmission()).

Синтаксис

```
Wire.write(value)
Wire.write(string)
Wire.write(data, length)
```

Параметры

value: значение, которое необходимо отправить в виде одиночного байта

string: строка, которую необходимо отправить в виде последовательности байт

data: массив данных, который необходимо отправить в виде нескольких байт

length: количество передаваемых байт

Возвращаемые значения

byte

Функция write() возвращает количество записанных байт. Считывание этого значения не обязательно

Пример

```
#include <Wire.h>

byte val = 0;

void setup()
{
  Wire.begin(); // подключаемся к шине i2c
}

void loop()
```

```
{
  Wire.beginTransmission(44); // начинаем процедуру передачи
устройству с адресом #44 (0x2c)
                              // адрес устройства указан в
даташите
  Wire.write(val);             // отправляем байт данных
  Wire.endTransmission();     // завершаем процедуру передачи

  val++;         // увеличиваем значение
  if(val == 64) // при достижении значения 64 (максимум)
  {
    val = 0;     // начинаем счет заново
  }
  delay(500);
}
```

Смотрите также

WireRead()
WireBeginTransmission()
WireEndTransmission()
Serial.write()

Wire.available()

Описание

Возвращает количество байт, доступных для считывания функцией read(). На ведущем устройстве (Master) данная функция должна вызываться после функции requestFrom(), а на ведомом (Slave) - внутри обработчика onReceive().

Функция available() является наследником вспомогательного класса Stream.

Параметры

нет

Возвращаемые значения

Количество байт, доступных для считывания.

Смотрите также

Wire.read()
Stream.available()

read()

Описание

Данная функция считывает байт данных, полученный ведущим устройством от ведомого (либо наоборот) в результате выполнения функции requestFrom(). Функция read() является наследником вспомогательного класса Stream.

Синтаксис

```
Wire.read()
```

Параметры

нет

Возвращаемые значения

Очередной полученный байт

Пример

```
#include <Wire.h>

void setup()
{
  Wire.begin();         // подключаемся к шине i2c (для ведущего устройства адрес не обязательный)
  Serial.begin(9600);  // инициализируем последовательный порт для вывода информации
}

void loop()
{
  Wire.requestFrom(2, 6);     // запрашиваем у ведомого устройства #2 6 байт

  while(Wire.available())     // ведомое устройство может отправить не все запрашиваемые байты
  {
    char c = Wire.read();     // считываем байт данных в виде символа
    Serial.print(c);          // выводим символ
  }
```

```
  delay(500);
}
```

Смотрите также

WireWrite()
WireAvailable()
WireRequestFrom()
Stream.read()

Wire.onReceive(handler)

Описание

На ведомом устройстве позволяет назначить функцию, которая будет автоматически вызываться при поступлении данных от Мастера.

Параметры

handler: функция, которую необходимо вызвать при поступлении данных от ведущего устройства; эта функция не должна возвращать никаких значений и может принимать только один параметр (int), описывающий количество поступивших байт. Например: void myHandler(int numBytes)

Возвращаемые значения

нет

Смотрите также

Wire.onRequest()

Wire.onRequest(handler)

Описание

На ведомом устройстве позволяет назначить функцию, которая будет автоматически вызываться при получения запроса от Мастера.

Параметры

handler: вызываемая функция без параметров, не должна возвращать никаких значений. Например: void myHandler()

Возвращаемые значения

нет

Смотрите также

Wire.onReceive()

Serial

Класс Serial используется для связи платы Ардуино с компьютером или другими устройствами. Все платы Arduino имеют, по крайней мере, один последовательный порт (также известный как UART или USART): **Serial**. Он связан с цифровыми выводами 0 (RX) и 1 (TX), а также используется для связи с компьютером через USB. Таким образом, во время использования последовательного порта, выводы 0 и 1 не могут использоваться в качестве цифровых входов или выходов.

Для связи с Arduino можно использовать специальную программу мониторинга последовательного порта, встроенную в программное обеспечение Ардуино. Для вызова программы нажмите соответствующую кнопку на панели инструментов и установите ту же скорость передачи, что указывается в вашей программе при вызове метода begin().

Arduino Mega имеет три дополнительных последовательных порта: **Serial1** с выводами 19 (RX) и 18 (TX), **Serial2** с выводами 17 (RX) и 16 (TX), **Serial3** с выводами 15 (RX) и 14 (TX). Данные выводы не связаны с преобразователем USB-UART на плате Mega, поэтому, для организации связи с компьютером через эти выводы понадобится дополнительный внешний преобразователь USB-UART. Для связи же с другим внешним устройством, имеющим последовательный TTL-порт, достаточно соединить всего три вывода: вывод TX с выводом RX устройства, вывод RX - с выводом TX устройства, а землю Mega, соответственно, с землей внешнего устройства. (Не подсоединяйте эти выводы к последовательному порту RS232 напрямую, поскольку последний работает с напряжениями +/- 12В и может повредить плату Ардуино.)

Arduino Due имеет три дополнительных последовательных порта с TTL-уровнем 3.3В: **Serial1** с выводами 19 (RX) и 18 (TX); **Serial2** с выводами 17 (RX) и 16 (TX), **Serial3** с выводами 15 (RX) и 14 (TX). Выводы 0 и 1 по-прежнему соединены с соответствующими выводами преобразователя интерфейсов USB-Serial TTL, реализованным на микросхеме ATmega16U2, связанной с отладочным портом USB. Помимо этого, благодаря возможностям микросхем SAM3X, на плате реализована и аппаратная поддержка порта USB, *SerialUSB*.

В Arduino Leonardo **Serial1** используется для связи через последовательный TTL-порт (5В) посредством выводов 0 (RX) и 1 (TX). **Serial** зарезервировано для USB CDC-связи. Для получения дополнительной информации см. страницы Leonardo - начало работы и описание платы.

if (Serial)

Описание

Позволяет проверить готовность определенного последовательного порта.

В Arduino Leonardo **if (Serial)** позволяет узнать, открыто ли USB CDC соединение. Во всех остальных случаях, результатом выполнения оператора **if (Serial1)** на Arduino Leonardo будет значение true.

Данная инструкция была введена в ARduino 1.0.1.

Синтаксис

Для всех плат:

```
if (Serial)
```

Только для Arduino Leonardo:

```
if (Serial1)
```

Только для Arduino Mega:

```
if (Serial1)
if (Serial2)
if (Serial3)
```

Параметры

нет

Возвращаемые значения

boolean: возвращает true, если указанный последовательный порт готов к работе. Инструкция может вернуть false только в том случае, если ее вызвать перед открытием USB CDC соединения на Arduino Leonardo.

Пример

```
void setup() {
 //Инициализируем последовательный интерфейс и ожидаем открытия порта:
  Serial.begin(9600);
  while (!Serial) {
    ; // ожидаем подключения последовательного порта. Нужно только для Leonardo
```

```
  }
}

void loop() {
 //продолжаем работу
}
```

Смотрите также

begin()
end()
available()
read()
peek()
flush()
print()
println()
write()
SerialEvent()

available()

Описание

Возвращает количество байт (символов) доступных для считывания из буфера последовательного порта. Под символами понимаются данные, которые уже приняты и хранятся в последовательном приемном буфере (который может хранить максимум 64 байта). Функция available() является наследником вспомогательного класса Stream.

Синтаксис

```
Serial.available()
```

Только для Arduino Mega:

```
Serial1.available()
Serial2.available()
Serial3.available()
```

Параметры

нет

Возвращаемые значения

количество байт, доступных для считывания

Пример

```
int incomingByte = 0;   // для данных, поступающих через
последовательный порт

void setup() {
        Serial.begin(9600);     // открываем последовательный
порт
                                //и задаем скорость обмена 9600
бод
}

void loop() {

        // отправляем данные только после их получения:
        if (Serial.available() > 0) {
                // считываем входящий байт:
```

```
                incomingByte = Serial.read();

                // показываем, что именно мы получили:
                Serial.print("I received: ");
                Serial.println(incomingByte, DEC);
        }
}
```

Пример для Arduino Mega:

```
void setup() {
  Serial.begin(9600);
  Serial1.begin(9600);

}

void loop() {
  // считываем с порта номер 0, отправляем на порт 1:
  if (Serial.available()) {
    int inByte = Serial.read();
    Serial1.print(inByte, BYTE);

  }
  // считываем с порта номер 1, отправляем на порт 0:
  if (Serial1.available()) {
    int inByte = Serial1.read();
    Serial.print(inByte, BYTE);
  }
}
```

Смотрите также

begin()
end()
available()
read()
peek()
flush()
print()
println()
write()
SerialEvent()
Stream.available()

begin()

Описание

Задает скорость передачи данных по последовательному интерфейсу в битах в секунду (бодах). Для взаимодействия с компьютером следует использовать одну из предустановленных скоростей обмена: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600 или 115200. Тем не менее, можно задавать и другие скорости - например, для взаимодействия с каким-либо специфичным компонентом посредством выводов 0 и 1.

Необязательный второй аргумент этой функции позволяет настроить количество бит данных, проверку четности и стоповые биты. По умолчанию, посылка состоит из 8 бит данных, без проверки четности, с одним стоповым битом.

Синтаксис

```
Serial.begin(speed)
Serial.begin(speed, config)
```

Только для Arduino Mega:

```
Serial1.begin(speed)
Serial2.begin(speed)
Serial3.begin(speed)
Serial1.begin(speed, config)
Serial2.begin(speed, config)
Serial3.begin(speed, config)
```

Параметры

speed: скорость в битах в секунду (бодах) - *long*
config: задает количество бит данных, проверку четности и стоповые биты. Ниже приведен список возможных значений:

SERIAL_5N1
SERIAL_6N1
SERIAL_7N1
SERIAL_8N1 (по умолчанию)
SERIAL_5N2
SERIAL_6N2
SERIAL_7N2
SERIAL_8N2

SERIAL_5E1
SERIAL_6E1
SERIAL_7E1
SERIAL_8E1
SERIAL_5E2
SERIAL_6E2
SERIAL_7E2
SERIAL_8E2
SERIAL_5O1
SERIAL_6O1
SERIAL_7O1
SERIAL_8O1
SERIAL_5O2
SERIAL_6O2
SERIAL_7O2
SERIAL_8O2

Возвращаемые значения

нет

Пример

```
void setup() {
    Serial.begin(9600); // открываем последовательный порт,
задаем скорость передачи данных 9600 бод
}

void loop() {}
```

Пример для Arduino Mega:

```
// В Arduino Mega используются все четыре последовательных порта
// (Serial, Serial1, Serial2, Serial3),
// с различной скоростью обмена данными:

void setup(){
  Serial.begin(9600);
  Serial1.begin(38400);
  Serial2.begin(19200);
  Serial3.begin(4800);

  Serial.println("Hello Computer");
  Serial1.println("Hello Serial 1");
  Serial2.println("Hello Serial 2");
```

```
  Serial3.println("Hello Serial 3");
}

void loop() {}
```

Смотрите также

begin()
end()
available()
read()
peek()
flush()
print()
println()
write()
SerialEvent()

end()

Описание

Функция разрывает последовательную связь, после чего выводы RX и TX снова могут использовать как выводы общего назначения. Для возобновления последовательного соединения используйте функцию Serial.begin().

Синтаксис

```
Serial.end()
```

Только для Arduino Mega:

```
Serial1.end()
Serial2.end()
Serial3.end()
```

Параметры

нет

Возвращаемые значения

нет

Смотрите также

begin()
end()
available()
read()
peek()
flush()
print()
println()
write()
SerialEvent()

Serial.find()

Описание

Функция Serial.find() осуществляет чтение данных из последовательного буфера до тех пор, пока не будет найдена искомая строка заданной длины. Функция возвращает true, если искомая строка найдена, и false - в случае таймаута.

Функция Serial.find() является наследником вспомогательного класса Stream.

Синтаксис

```
Serial.find(target)
```

Параметры

target : искомая строка (char)

Возвращаемые значения

boolean

Смотрите также

Stream
Stream.find()

Serial.findUntil()

Описание

Функция Serial.findUntil() считывает данные из последовательного буфера до тех пор, пока не будет найдена искомая строка определенной длины или символы конца строки.

Функция возвращает true, если искомая строка найдена, и false - в случае таймаута.

Функция Serial.findUntil() является наследником вспомогательного класса Stream.

Синтаксис

```
Serial.findUntil(target, terminal)
```

Параметры

target : искомая строка (char)
terminal : символы конца строки (char)

Возвращаемые значения

boolean

Смотрите также

Stream
Stream.findUntil()

flush()

Описание

Функция ожидает завершения процесса отправки всех исходящих данных по последовательному интерфейсу. (В более ранних версиях Ардуино - до версии 1.0 - эта функция просто очищала буфер приема от поступивших в него данных).

Функция flush() является наследником вспомогательного класса Stream.

Синтаксис

```
Serial.flush()
```

Только для Arduino Mega:

```
Serial1.flush()
Serial2.flush()
Serial3.flush()
```

Параметры

нет

Возвращаемые значения

нет

Смотрите также

begin()
end()
available()
read()
peek()
flush()
print()
println()
write()
SerialEvent()
Stream.flush()

Serial.parseFloat()

Описание

Функция Serial.parseFloat() анализирует последовательный буфер и возвращает первое попавшееся число с плавающей точкой. В процессе анализа буфера функцией игнорируются знак "минус", а также все не цифровые символы. parseFloat() завершает свою работу при обнаружении первого символа, не относящегося к искомому дробному числу.

Функция Serial.parseFloat() является наследником вспомогательного класса Stream.

Синтаксис

```
Serial.parseFloat()
```

Параметры

нет

Возвращаемые значения

float

Смотрите также

Stream
Stream.parseFloat()

parseInt()

Описание

Осуществляет поиск очередного целого числа во входном потоке. Функция parseInt() является наследником вспомогательного класса Stream.

Если в течение одной секунды не найдено ни одного целого числа, то, по умолчанию, функция вернет значение 0. Величина таймаута задается функцией Serial.setTimeout().

Синтаксис

```
Serial.parseInt()
```

Только для Arduino Mega:

```
Serial1.parseInt()
Serial2.parseInt()
Serial3.parseInt()
```

Параметры

нет

Возвращаемые значения

int : найденное целое число

Смотрите также

begin()
end()
available()
read()
peek()
flush()
print()
println()
write()
SerialEvent()
Stream.parseInt()

peek()

Описание

Возвращает очередной байт (символ), поступивший в буфер последовательного порта. При этом функция не удаляет считанный байт из буфера. Таким образом, при многократном вызове peek() функция будет возвращать одно и то же значение, как и функция read(). peek() является наследником вспомогательного класса Stream.

Синтаксис

```
Serial.peek()
```

Только для Arduino Mega:

```
Serial1.peek()
Serial2.peek()
Serial3.peek()
```

Параметры

нет

Возвращаемые значения

первый байт данных, доступный для чтения из буфера последовательного порта (либо -1, если таковой байт отсутствует) - *int*.

Смотрите также

begin()
end()
available()
read()
peek()
flush()
print()
println()
write()
SerialEvent()
Stream.peek()

print()

Описание

Функция выводит через последовательный порт заданный ASCII-текст в виде, понятном для человека. Эта команда может иметь несколько разных форм. При выводе числа каждой его цифре соответствует один ASCII-символ. Дробные числа тоже выводятся в виде ASCII-цифр, при этом после запятой по умолчанию оставляется два десятичных знака. Байты выводятся в виде отдельных символов, а символы и строки выводятся без изменений - "как есть". Например:

Serial.print(78) - выведет "78"
Serial.print(1.23456) - выведет "1.23"
Serial.print('N') - выведет "N"
Serial.print("Hello world.") - выведет "Hello world."

Необязательный второй параметр задает формат вывода; этот параметр может принимать следующие значения: BIN (двоичная система с основанием 2), OCT (восьмеричная система с основанием 8), DEC (десятичная система с основанием 10), HEX (шестнадатиричная система с основанием 16). Для числе с плавающей точкой этот параметр определяет количество десятичных знаков после запятой. Например:

Serial.print(78, BIN) - выведет "1001110"
Serial.print(78, OCT) - выведет "116"
Serial.print(78, DEC) - выведет "78"
Serial.print(78, HEX) - выведет "4E"
Serial.println(1.23456, 0) - выведет "1"
Serial.println(1.23456, 2) - выведет "1.23"
Serial.println(1.23456, 4) - выведет "1.2346"

Функции Serial.print() можно передавать и строки, записанные во флеш-память контроллера. Для этого их нужно заключить в конструкцию F(). Например:

Serial.print(F(“Hello World”))

Для отправки отдельного байта используйте функцию Serial.write().

Синтаксис

```
Serial.print(val)
Serial.print(val, format)
```

Параметры

val: значение, которое необходимо вывести - любой тип данных

format: определяет систему счисления (для целочисленных типов), а также количество десятичных знаков после запятой (для чисел с плавающей точкой).

Возвращаемые значения

size_t (long): функция print() возвращает количество выведенных байт. Считывание этого значения не обязательно.

Пример

```
/*
Использования цикла FOR для обработки данных и вывода чисел в
различных форматах.
*/
int x = 0;     // переменная

void setup() {
  Serial.begin(9600);       // открываем последовательный порт на
скорости 9600 бод
}

void loop() {
  // print labels
  Serial.print("NO FORMAT");        // выводим метку
  Serial.print("\t");               // выводим символ табуляции

  Serial.print("DEC");
  Serial.print("\t");

  Serial.print("HEX");
  Serial.print("\t");

  Serial.print("OCT");
  Serial.print("\t");

  Serial.print("BIN");
  Serial.print("\t");

  for(x=0; x< 64; x++){     // немного ASCII-графики, можете
изменить код по своему вкусу

    // выводим значение в различных форматах:
    Serial.print(x);        // выводим как десятичное число - то
же, что и формат "DEC"
    Serial.print("\t");     // выводим символ табуляции
```

```
    Serial.print(x, DEC);  // выводим десятичное число ASCII-
символами
    Serial.print("\t");    // выводим символ табуляции

    Serial.print(x, HEX);  // выводим шестнадатиричное число
ASCII-символами
    Serial.print("\t");    // выводим символ табуляции

    Serial.print(x, OCT);  // выводим восьмиричное число ASCII-
символами
    Serial.print("\t");    // выводим символ табуляции

    Serial.println(x, BIN);  // выводим двоичное число ASCII-
символами
    //с символом возврата каретки, который добавляет функция
"println"
    delay(200);            // задержка в 200 миллисекунд
  }
  Serial.println("");      // выводим еще один символ возврата
каретки
}
```

Совет

Во всех версиях Ардуино, начиная с версии 1.0, последовательная передача данных осуществляется асинхронно, вследствие чего функция Serial.print() завершается до самой отправки данных.

Смотрите также

begin()
end()
available()
read()
peek()
flush()
print()
println()
write()
SerialEvent()
Memory

println()

Описание

Выводит через последовательный порт ASCII-текст в понятном для человека виде с символами возврата каретки (ASCII 13 или '\r') и новой строки (ASCII 10 или 'n'). Эта команда имеет такие же формы, как и Serial.print().

Синтаксис

```
Serial.println(val)
Serial.println(val, format)
```

Параметры

val: значение, которое необходимо вывести - любой тип данных

format: определяет систему счисления (для целочисленных типов), а также количество десятичных знаков после запятой (для чисел с плавающей точкой).

Возвращаемые значения

size_t (long): функция println() возвращает количество выведенных байт. Считывание этого значения не обязательно.

Пример

```
/*
  Аналоговый вход

 считываем аналоговое значение с аналогового входа 0 и выводим
его через последовательный порт.

 создано 24 марта 2006
 Автор - Tom Igoe
 */

int analogValue = 0;    // переменная для хранения аналоговой
величины

void setup() {
  // открываем последовательный порт на скорость 9600 бод:
  Serial.begin(9600);
}
```

```
void loop() {
  // считываем аналговую величиниу на выводе 0:
  analogValue = analogRead(0);

  // выводим считанное значение в различных форматах:
  Serial.println(analogValue);       // выводим десятичное число
ASCII-символами
  Serial.println(analogValue, DEC);  // выводим десятичное число
ASCII-символами
  Serial.println(analogValue, HEX);  // выводим шестнадатиричное
число ASCII-символами
  Serial.println(analogValue, OCT);  // выводим восьмиричное
число ASCII-символами
  Serial.println(analogValue, BIN);  // выводим двоичное число
ASCII-символами

  // пауза в 10 миллисекунд перед следующим опросом:
  delay(10);
}
```

Смотрите также

begin()
end()
available()
read()
peek()
flush()
print()
println()
write()
SerialEvent()

read()

Описание

Считывает данные, поступающие по последовательному интерфейсу. Функция read() является наследником вспомогательного класса Stream.

Синтаксис

Serial.read()

Только для Arduino Mega:

Serial1.read()
Serial2.read()
Serial3.read()

Параметры

нет

Возвращаемые значения

Первый байт принятых данных (или -1, если таковых нет) - *int*

Пример

```
int incomingByte = 0;    // переменная для хранения байта данных,
                         //принятых по последовательному
интерфейсу

void setup() {
        Serial.begin(9600);     // открываем последовательный
порт
                                // задаем скорость передачи
данных 9600 бод
}

void loop() {

        // отправляем данные только после их получения:
        if (Serial.available() > 0) {
                // считываем принятый байт:
                incomingByte = Serial.read();
```

```
                // сообщаем, что именно мы получили:
                Serial.print("I received: ");
                Serial.println(incomingByte, DEC);
        }
}
```

Смотрите также

begin()
end()
available()
read()
peek()
flush()
print()
println()
write()
SerialEvent()
Stream.read()

Serial.readBytes()

Описание

Функция Serial.readBytes() считывает символы, поступающие через последовательный порт, и помещает их в приемный буфер. Функция прекращает свою работу после обработки заданного количества символов, либо в случае таймаута (см. Serial.setTimeout()).

Функция возвращает количество байт, помещенных в буфер, либо 0 - если таковых нет.

Функция Serial.readBytes() является наследником вспомогательного класса Stream.

Синтаксис

```
Serial.readBytes(buffer, length)
```

Параметры

buffer: буфер для хранения считываемых данных (char[] или byte[])
length : количество символов, которое необходимо считать (int).

Возвращаемые значения

byte

Смотрите также

Stream
Stream.readBytes()

Serial.readBytesUntil()

Описание

Функция Serial.readBytesUntil() считывает символы из приемного буфера последовательного порта и помещает их в массив. Функция прекращает свою работу после обработки заданного количества символов, в случае обнаружения указанного символа, либо в случае таймаута (см. Serial.setTimeout()).

Функция возвращает количество байт, помещенных в буфер, либо 0 - если таковых нет.

Функция Serial.readBytesUntil() является наследником вспомогательного класса Stream.

Синтаксис

```
Serial.readBytesUntil(character, buffer, length)
```

Параметры

character: искомый символ (char).

buffer: буфер для хранения считываемых данных (char[] или byte[])

length: количество символов, которое необходимо считать (int).

Возвращаемые значения

byte

Смотрите также

Stream
Stream.readBytesUntil()

Serial.setTimeout()

Описание

Функция setTimeout() позволяет задать время ожидания данных, поступающих через последовательный интерфейс. Таймаут задается в миллисекундах (по умолчанию 1000 мс) и используется в функциях Serial.readBytesUntil() и Serial.readBytes().

Функция Serial.setTimeout() является наследником вспомогательного класса Stream.

Синтаксис

```
Serial.setTimeout(time)
```

Параметры

time: величина таймаута в миллисекундах (long).

Возвращаемые значения

нет

Смотрите также

Stream
Stream.setTimeout()

write()

Описание

Записывает двоичные данные в последовательный порт. Эти данные отправляются в виде одного или нескольких байтов; для отправки символов, представляющих собой цифры какого-либо числа, используйте функцию print ().

Синтаксис

```
Serial.write(val)
Serial.write(str)
Serial.write(buf, len)
```

Arduino Mega также поддерживает:

```
Serial1, Serial2, Serial3
```

Параметры

val: значение, которое необходимо отправить в виде байта

str: строка, которую необходимо отправить как последовательность байт

buf: массив, который необходимо отправить как последовательность байт

len: длина массива

Возвращаемые значения

byte
Функция write() возвращает количество отправленных байт. Считывание этого параметра не обязательно.

Пример

```
void setup(){
  Serial.begin(9600);
}

void loop(){
  Serial.write(45); // отправляем байт со значением 45

   int bytesSent = Serial.write(“hello”); //отправляем строку
"hello” и возвращаем длину строки.
```

```
}
```

Смотрите также

[begin()](#)
[end()](#)
[available()](#)
[read()](#)
[peek()](#)
[flush()](#)
[print()](#)
[println()](#)
[write()](#)
[SerialEvent()](#)

serialEvent()

Описание

Автоматически вызывается при поступлении новых данных. Чтобы считать их, используйте функцию Serial.read().

Примечание: в настоящее время функция serialEvent() не совместима с Arduino Esplora, Leonardo и Micro.

Синтаксис

```
void serialEvent(){
//команды
}
```

Только для Arduino Mega:

```
void serialEvent1(){
//команды
}

void serialEvent2(){
//команды
}

void serialEvent3(){
//команды
}
```

Параметры

команды - любые процедуры и функции Ардуино

Смотрите также

SerialEvent Tutorial
begin()
end()
available()
read()
peek()
flush()
print()

println()
write()
SerialEvent()

Stream

Класс Stream является базовым классом для символьных и двоичных потоков. Как правило, этот класс не используется напрямую - он вызывается при использовании функций, основанных на нем.

В классе Stream определены функции чтения Ардуино. Поэтому, при вызове любых базовых функций, использующих методы подобные read(), можно с уверенностью подразумевать обращение к классу Stream. Для функций подобных print(), Stream является потомком класса Print.

К библиотекам, использующим класс Stream, относятся:

Serial
Wire
Ethernet Client
Ethernet Server
SD

available()

Описание

Функция available() возвращает количество принятых, но не прочитанных байт из потока.

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.available()
```

Параметры

stream: экземпляр класса, наследуемого от Stream.

Возвращаемые значения

int: количество непрочитанных байт

Смотрите также

Stream

read()

Описание

Функция read() осуществляет считывание символов из входящего потока в указанный буфер.

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.read()
```

Параметры

stream: экземпляр класса, наследуемого от Stream.

Возвращаемые значения

первый байт принятых данных (или -1, если таковых нет)

Смотрите также

Stream()

flush()

Описание

Функция flush() очищает буфер сразу после отправки всех исходящих символов.

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.flush()
```

Параметры

stream: экземпляр класса, наследуемого от Stream.

Возвращаемые значения

boolean

Смотрите также

Stream()

find()

Описание

Функция find() считывает из потока данные до тех пор, пока не будет найдена искомая строка заданной длины. Функция возвращает true, если искомая строка найдена, false - в случае таймаута.

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.find(target)
```

Параметры

stream: экземпляр класса, наследуемого от Stream.

target: искомая строка (char)

Возвращаемые значения

boolean

Смотрите также

Stream()

findUntil()

Описание

Функция findUntil() осуществляет поиск строки в потоке данных. В процессе поиска считывание данных из потока осуществляется до тех пор, пока не будет найдена искомая строка заданной длины или символы конца строки.

Функция возвращает true, если искомая строка найдена, false - в случае таймаута.

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.findUntil(target, terminal)
```

Параметры

stream: экземпляр класса, наследуемого от Stream.

target: искомая строка (char)

terminal: символы конца строки (char)

Возвращаемые значения

boolean

Смотрите также

Stream()

peek()

Описание

Считывает из файла байт данных. При этом функция сохраняет положение указателя текущей позиции файла и не перемещает его на следующий байт. Т.е. при многократном вызове peek() функция будет возвращать одно и то же значение.

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.peek()
```

Параметры

stream: экземпляр класса, наследуемого от Stream.

Возвращаемые значения

Байт данных (или символ), либо -1, если таковой отсутствует.

Смотрите также

available()
read()

readBytes()

Описание

Функция readBytes() считывает символы из потока и помещает их в указанный буфер. Функция прекращает свою работу после обработки заданного количества символов, либо в случае таймаута (см. setTimeout()).

Функция возвращает количество байт, помещенных в буфер, либо 0 - если таковых нет.

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.readBytes(buffer, length)
```

Параметры

stream: экземпляр класса, наследуемого от Stream.

buffer: буфер для хранения считываемых данных (char[] или byte[])

length: количество символов, которое необходимо считать (int).

Возвращаемые значения

Количество байт, помещенных в буфер

Смотрите также

Stream

readBytesUntil()

Описание

Функция readBytesUntil() считывает символы из потока и помещает их в указанный буфер. Функция прекращает свою работу после обработки заданного количества символов, в случае обнаружения указанного символа, либо в случае таймаута (см. setTimeout()).

readBytesUntil() возвращает количество байт, помещенных в буфер, либо 0 - если таковых нет.

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.readBytesUntil(character, buffer, length)
```

Параметры

stream: экземпляр класса, наследуемого от Stream.

character: искомый символ (char).

buffer: буфер для хранения считываемых данных (char[] или byte[])

length: количество символов, которое необходимо считать (int).

Возвращаемые значения

Количество байт, помещенных в буфер

Смотрите также

Stream

readString()

Описание

Функция readString() считывает символы из потока, объединяя их в строку. Функция прекращает свою работу в случае таймаута (см. setTimeout()).

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.readString()
```

Параметры

нет

Возвращаемые значения

Строка, состоящая из считанных символов.

Смотрите также

Stream

readStringUntil()

Описание

Функция readStringUntil() считывает символы из потока, объединяя их в строку. Функция прекращает свою работу в случае обнаружения указанного символа, либо в случае таймаута (см. setTimeout()).

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.readString(terminator)
```

Параметры

terminator: искомый символ (char).

Возвращаемые значения

Строка, состоящая из считанных символов.

Смотрите также

Stream

parseInt()

Описание

Функция parseInt() осуществляет поиск целого числа во входном потоке. Процесс поиска заключается в следующем: начиная с текущей позиции, функция анализирует символы входного потока до тех пор, пока не встретится первое числовое значение. При этом все остальные символы (включая знак минус) функцией игнорируются. После нахождения числа функция parseInt() завершает свою работу при обнаружении первого не цифрового символа.

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.parseInt(list)
```

Параметры

stream: экземпляр класса, наследуемого от Stream.

list: анализируемый поток, в котором необходимо найти целое число (char)

Возвращаемые значения

Первое попавшееся целое число (int)

Смотрите также

Stream

parseFloat()

Описание

Функция parseFloat() осуществляет поиск во входном потоке числа с плавающей точкой. Процесс поиска заключается в следующем: начиная с текущей позиции, функция анализирует символы входного потока до тех пор, пока не встретится первое числовое значение. При этом все остальные символы (включая знак минус) функцией игнорируются. После нахождения искомого числа функция parseFloat() завершает свою работу при обнаружении первого не цифрового символа.

Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.parseFloat(list)
```

Параметры

stream: экземпляр класса, наследуемого от Stream.

list: анализируемый поток, в котором необходимо найти число с плавающей точкой (char)

Возвращаемые значения

Первое попавшееся число с плавающей точкой (float)

Смотрите также

Stream

setTimeout()

Описание

Функция setTimeout() устанавливает максимальное время ожидания потока данных (значение по умолчанию - 1000 миллисекунд). Данная функция является одним из методов класса Stream, поэтому вызывается любым объектом, являющимся наследником этого класса (Wire, Serial и др.). Для получения дополнительной информации см. описание класса Stream.

Синтаксис

```
stream.setTimeout(time)
```

Параметры

stream: экземпляр класса, наследуемого от Stream.

time: величина таймаута в миллисекундах (long).

Возвращаемые значения

нет

Смотрите также

Stream

Библиотеки Mouse и Keyboard

Эти базовые библиотеки позволяют платам Arduino Leonardo, Micro или Due при подключении к компьютеру определяться как обычная мышь и/или клавиатура.

Предосторожности при использовании библиотек Mouse и Keyboard: при постоянной работе библиотек Mouse или Keyborad могут возникнуть сложности во время программирования вашего устройства. Функции, подобные Mouse.move() или Keyboard.print() могут перемещать курсор и сигнализировать о нажатии клавиш подключенному компьютеру, поэтому должны вызываться только тогда, когда вы готовы контролировать их работу. Рекомендуется использовать какую-нибудь систему управления, позволяющую выключать подобную функциональность, например, в зависимости от положения переключателя или сигнала на выводе, состояние которого вы можете контролировать.

Перед использованием функций библиотек Mouse или Keyboard непосредственно с компьютером, лучше всего протестировать их возвращаемые значения с помощью функции Serial.print(). Так вы сможете удостовериться в правильности отправляемых значений. Подробнее см. примеры работы с библиотеками Mouse и Keyboard ниже.

Мышь (библиотека Mouse)

Функции для работы с мышью позволяют Leonardo, Micro или Due контролировать движение курсора на подключенном компьютере. Обновление позиции курсора всегда осущетсвляется относительно его предыдущего положения.

Mouse.begin()
Mouse.click()
Mouse.end()
Mouse.move()
Mouse.press()
Mouse.release()
Mouse.isPressed()

Клавиатура (библиотека Keyboard)

Функции для работы с клавиатурой позволяют Leonardo, Micro или Due отправлять подключенному компьютеру сигналы о нажатии клавиш.

Примечание: Библиотека Keyboard позволяет отправлять не все ASCII-символы, в частности она не позволяет отправлять непечатаемые символы. Библиотека также поддерживает использование клавиш-модификаторов, которые при одновременном нажатии с другой клавишей изменяют ее поведение. Дополнительную информацию о поддерживаемых клавишах и их применении см. здесь.

Keyboard.begin()
Keyboard.end()
Keyboard.press()
Keyboard.print()
Keyboard.println()
Keyboard.release()
Keyboard.releaseAll()
Keyboard.write()

Mouse.begin()

Описание

Метод заставляет Ардуино начать эмулировать мышь, подсоединенную к компьютеру. Метод begin() должен вызываться до начала управления компьютером. Для завершения эмуляции используйте метод Mouse.end().

Синтаксис

```
Mouse.begin()
```

Параметры

нет

Возвращаемые значения

нет

Пример

```
void setup(){
 pinMode(2, INPUT);
}

void loop(){

  //инициализируем библиотеку Mouse при нажатии кнопки
  if(digitalRead(2) == HIGH){
     Mouse.begin();
   }

}
```

Смотрите также

Mouse.click()
Mouse.end()
Mouse.move()
Mouse.press()
Mouse.release()
Mouse.isPressed()

Mouse.click()

Описание

Отправляет компьютеру сигнал о щелчке кнопкой мыши в текущей позиции курсора. Щелчок подразумевает мгновенное нажатие и отпускание кнопки.

По умолчанию Mouse.click() сигнализирует о нажатии левой кнопки мыши.

ВНИМАНИЕ: при использовании команды Mouse.click() Ардуино берет управление вашей мышью на себя! Поэтому, перед использованием этой команды убедитесь, что у вас есть возможность отключить Ардуино от управления мышью. С этой целью рекомендуется использовать какую-либо кнопку, позволяющую включать или выключать данную функцию.

Синтаксис

```
Mouse.click();
Mouse.click(button);
```

Параметры

button: характеризует кнопку мыши, сигнал нажатия которой будет отправлен компьютеру - *char*

MOUSE_LEFT (по умолчанию)
MOUSE_RIGHT
MOUSE_MIDDLE

Возвращаемые значения

нет

Пример

```
void setup(){
  pinMode(2,INPUT);
  //инициализируем библиотеку Mouse
  Mouse.begin();
}

void loop(){
  //при нажатии кнопки отправляем сигнал нажатия правой кнопкой мыши
  if(digitalRead(2) == HIGH){
    Mouse.click();
```

```
    }
}
```

Смотрите также

Mouse.begin()
Mouse.end()
Mouse.move()
Mouse.press()
Mouse.release()
Mouse.isPressed()

Mouse.end()

Описание

Заставляет Ардуино прекратить эмуляцию подсоединенной к компьютеру мыши. Для начала эмуляции используйте метод Mouse.begin().

Синтаксис

```
Mouse.end()
```

Параметры

нет

Возвращаемые значения

нет

Пример

```
void setup(){
  pinMode(2,INPUT);
  //инициализируем библиотеку Mouse
  Mouse.begin();
}

void loop(){
  //при нажатии кнопки отправляем сигнал нажатия правой кнопки мыши,
  //после чего завершаем процесс эмуляции мыши.
  if(digitalRead(2) == HIGH){
    Mouse.click();
    Mouse.end();
  }
}
```

Смотрите также

Mouse.begin()
Mouse.click()
Mouse.move()
Mouse.press()
Mouse.release()

Mouse.isPressed()

Mouse.move()

Описание

Перемещает указатель мыши на подсоединенном компьютере. Перемещения по экрану всегда задаются относительно текущего положения указателя. Перед использованием Mouse.move() необходимо вызвать метод Mouse.begin().

ВНИМАНИЕ: при использовании команды Mouse.move() Ардуино берет управление вашей мышью на себя! Поэтому, перед использованием этой команды убедитесь, что у вас есть возможность отключить Ардуино от управления мышью. С этой целью рекомендуется использовать какую-либо кнопку, позволяющую включать или выключать данную функцию.

Синтаксис

```
Mouse.move(xVal, yVal, wheel)
```

Параметры

xVal: величина, на которую следует переместить указатель вдоль оси x - *signed char*

yVal: величина, на которую следует переместить указатель вдоль оси y - *signed char*

wheel: величина смещения колеса прокрутки - *signed char*

Возвращаемые значения

нет

Пример

```
const int xAxis = A1;          // аналоговый датчик для оси X
const int yAxis = A2;          // аналоговый датчик для оси Y

int range = 12;                // диапазон перемещения по оси X и
Y
int responseDelay = 2;         // задержка отклика мыши в мс
int threshold = range/4;       // величина порога "зоны покоя",
при превышении которого считанное значение
                               // будет считаться перемещением
мыши
```

```
int center = range/2;          // центральная позиция "зоны покоя"
int minima[] = {
  1023, 1023};                 // фактический минимум,
возвращаемый функцией analogRead для осей {x, y}
int maxima[] = {
  0,0};                        // фактический максимум,
возвращаемый функцией analogRead для осей {x, y}
int axis[] = {
  xAxis, yAxis};               // номера выводов для {x, y}
соответственно
int mouseReading[2];           // итоговая величина перемещения
мыши для осей {x, y}


void setup() {
 Mouse.begin();
}

void loop() {

// считываем величину перемещения по двум осям:
  int xReading = readAxis(0);
  int yReading = readAxis(1);

// перемещаем указатель мыши:
    Mouse.move(xReading, yReading, 0);
    delay(responseDelay);
}

/*
  функция считывает величину перемещения по указанной оси (0 для
x, 1 для y)
  и преобразовывает считанное значение из входного диапазона в
диапазон перемещений
*/

int readAxis(int axisNumber) {
  int distance = 0;    // величина перемещения от центра
диапазона

  // считываем значение с аналогового входа:
  int reading = analogRead(axis[axisNumber]);

  // если текущее считанное значение превышает значения max или
min данной оси,
  // то переприсваиваем max или min:
```

```
  if (reading < minima[axisNumber]) {
    minima[axisNumber] = reading;
  }
  if (reading > maxima[axisNumber]) {
    maxima[axisNumber] = reading;
  }

  // преобразовываем считанное значение из входного диапазона в
диапазон перемещения:
  reading = map(reading, minima[axisNumber], maxima[axisNumber],
0, range);

  // если пересчитанное значение выходит за пределы "зоны
покоя",
  // то используем его в качестве величины перемещения:
  if (abs(reading - center) > threshold) {
    distance = (reading - center);
  }

  // для корректного перемещения по вертикали
  // значение по оси Y должно быть проинвертировано
  if (axisNumber == 1) {
    distance = -distance;
  }

  // возвращаем величину перемещения для данной оси:
  return distance;
}
```

Смотрите также

Mouse.begin()
Mouse.click()
Mouse.end()
Mouse.press()
Mouse.release()
Mouse.isPressed()

Mouse.press()

Описание

Отправляет компьютеру сигнал о нажатии кнопки мыши. При этом сигнал нажатия подразумевает нажатие и продолжительное удерживание кнопки. Отмена нажатия осуществляется с помощью метода Mouse.release().

Перед использованием Mouse.press() необходимо начать процесс эмуляции мыши с помощью команды Mouse.begin().

По умолчанию Mouse.press() сигнализирует о нажатии левой кнопки мыши.

ВНИМАНИЕ: при использовании команды Mouse.press() Ардуино берет управление вашей мышью на себя! Поэтому, перед использованием этой команды убедитесь, что у вас есть возможность отключить Ардуино от управления мышью. С этой целью рекомендуется использовать какую-либо кнопку, позволяющую включать или выключать данную функцию.

Синтаксис

```
Mouse.press();
Mouse.press(button);
```

Параметры

button: характеризует кнопку мыши, сигнал нажатия которой будет отправлен компьютеру - *char*

MOUSE_LEFT (по умолчанию)
MOUSE_RIGHT
MOUSE_MIDDLE

Возвращаемые значения

нет

Пример

```
void setup(){
  //переключатель, инициирующий нажатие кнопки мыши
  pinMode(2,INPUT);
  //переключатель, который будет прерывать нажатие кнопки
  pinMode(3,INPUT);
  //инициализируем библиотеку Mouse
  Mouse.begin();
```

```
}

void loop(){
  //если замкнут ключ, подсоединенный к выводу 2, то нажимаем и
удерживаем правую кнопку мыши
  if(digitalRead(2) == HIGH){
    Mouse.press();
  }
  //если замкнут ключ, подсоединенный к выводу 3, то отпускаем
правую кнопку мыши
  if(digitalRead(3) == HIGH){
    Mouse.release();
  }
}
```

Смотрите также

Mouse.begin()
Mouse.click()
Mouse.end()
Mouse.move()
Mouse.release()
Mouse.isPressed()

Mouse.release()

Описание

Отправляет сообщение о том, что нажатая ранее (с помощью Mouse.press()) кнопка мыши отпущена. По умолчанию Mouse.release() сигнализирует об освобождении левой кнопки мыши.

ВНИМАНИЕ: при использовании команды Mouse.release() Ардуино берет управление вашей мышью на себя! Поэтому, перед использованием этой команды убедитесь, что у вас есть возможность отключить Ардуино от управления мышью. С этой целью рекомендуется использовать какую-либо кнопку, позволяющую включать или выключать данную функцию.

Синтаксис

```
Mouse.release();
Mouse.release(button);
```

Параметры

button: характеризует кнопку мыши, сигнал об отпускании которой необходимо отправить компьютеру - *char*

MOUSE_LEFT (по умолчанию)
MOUSE_RIGHT
MOUSE_MIDDLE

Возвращаемые значения

нет

Пример

```
void setup(){
  //переключатель, инициирующий нажатие кнопки мыши
  pinMode(2,INPUT);
  //переключатель, который будет прерывать нажатие кнопки
  pinMode(3,INPUT);
  //инициализируем библиотеку Mouse
  Mouse.begin();
}

void loop(){
  //если замкнут ключ, подсоединенный к выводу 2, то нажимаем и
```

```
удерживаем правую кнопку мыши
  if(digitalRead(2) == HIGH){
    Mouse.press();
  }
  //если замкнут ключ, подсоединенный к выводу 3, то отпускаем
правую кнопку мыши
  if(digitalRead(3) == HIGH){
    Mouse.release();
  }
}
```

Смотрите также

Mouse.begin()
Mouse.click()
Mouse.end()
Mouse.move()
Mouse.press()
Mouse.isPressed()

Mouse.isPressed()

Описание

Проверяет текущее состояние всех кнопок мыши и возвращает true, если одна из них нажата.

Синтаксис

```
Mouse.isPressed();
Mouse.isPressed(button);
```

Параметры

Если параметр не указан, функция проверяет состояние левой кнопки мыши.

button: характеризует кнопку мыши, состояние которой необходимо проверить - *char*

MOUSE_LEFT (по умолчанию)
MOUSE_RIGHT
MOUSE_MIDDLE

Возвращаемые значения

boolean: возвращает true, если кнопка нажата, false - если отпущена.

Пример

```
void setup(){
  //переключатель, инициирующий нажатие кнопки мыши
  pinMode(2,INPUT);
  //переключатель, прерывающий нажатие кнопки мыши
  pinMode(3,INPUT);
  //инициализируем последовательную связь с компьютером
  Serial1.begin(9600);
  //инициализируем библиотеку Mouse
  Mouse.begin();
}

void loop(){
  //переменная для хранения состояния кнопки
  int mouseState=0;
  //если замкнут переключатель, подсоединенный к выводу 2, то
нажимаем и удерживаем правую
  //кнопку мыши, после чего сохраняем ее состояние в переменной
```

```
  if(digitalRead(2) == HIGH){
    Mouse.press();
    mouseState=Mouse.isPressed();
  }
  //если замкнут переключатель, подсоединенный к выводу 3, то
отпускаем правую кнопку мыши,
  //после чего сохраняем ее состояние в переменной
  if(digitalRead(3) == HIGH){
    Mouse.release();
    mouseState=Mouse.isPressed();
  }
  //выводим текущее состояние кнопки мыши
  Serial1.println(mouseState);
  delay(10);
}
```

Смотрите также

Mouse.begin()
Mouse.click()
Mouse.end()
Mouse.move()
Mouse.press()
Mouse.release()

Keyboard.begin()

Описание

Использование метода Keyboard.begin() на платах Leonardo или Due заставляет Ардуино начать эмулировать клавиатуру, подсоединенную к компьютеру. Для завершения эмуляции служит команда Keyboard.end().

Синтаксис

```
Keyboard.begin()
```

Параметры

нет

Возвращаемые значения

нет

Пример

```
void setup() {
  // переводим вывод 2 в режим входа и включаем
  // подтягивающий резистор, в результате чего на выводе
присутствует высокий уровень
  // до тех пор, пока он не будет соединен с землей:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //если кнопка нажата
  if(digitalRead(2)==LOW){
    //отправляем сообщение
    Keyboard.print("Hello!");
  }
}
```

Смотрите также

Keyboard.end()
Keyboard.press()
Keyboard.print()
Keyboard.println()

Keyboard.release()
Keyboard.releaseAll()
Keyboard.write()

Keyboard.end()

Описание

Заставляет Leonardo прекратить эмуляцию подсоединенной к компьютеру клавиатуры. Для возобновления эмуляции служит команда Keyboard.begin().

Синтаксис

```
Keyboard.end()
```

Параметры

нет

Возвращаемые значения

нет

Пример

```
void setup() {
  //начинаем эмуляцию клавиатуры
  Keyboard.begin();
  //отправляем сигналы нажатия клавиш
  Keyboard.print("Hello!");
  //завершаем эмуляцию клавиатуры
  Keyboard.end();
}

void loop() {
  //ничего не делаем
}
```

Смотрите также

Keyboard.begin()
Keyboard.press()
Keyboard.print()
Keyboard.println()
Keyboard.release()
Keyboard.releaseAll()
Keyboard.write()

Keyboard.press()

Описание

Keyboard.press() эмулирует нажатие и удерживание какой-либо клавиши. Данная команда удобна при эмуляции нажатий клавиш-модификаторов. Отмена нажатия осуществляется с помощью метода Keyboard.release() или Keyboard.releaseAll().

Метод press() необходимо использовать только после вызова Keyboard.begin().

Синтаксис

```
Keyboard.press()
```

Параметры

char: клавиша, нажатие которой необходимо эмулировать

Возвращаемые значения

нет

Пример

```
// для OSX используйте этот параметр:
char ctrlKey = KEY_LEFT_GUI;
// для Windows и Linux используйте этот параметр:
//  char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // переводим вывод 2 в режим входа и включаем
  // подтягивающий резистор, в результате чего на выводе
присутствует высокий уровень
  // до тех пор, пока он не будет соединен с землей:
  pinMode(2, INPUT_PULLUP);
  // инициализируем контроль над клавиатурой:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
    // ничего не делаем до тех пор, пока на выводе 2 не появится
низкий уровень
    delay(500);
```

```
  }
  delay(1000);
  // новый документ:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.releaseAll();
  // ждем открытия нового окна:
  delay(1000);
}
```

Смотрите также

Keyboard.begin()
Keyboard.end()
Keyboard.print()
Keyboard.println()
Keyboard.release()
Keyboard.releaseAll()
Keyboard.write()

Keyboard.print()

Описание

Отправляет подсоединенному компьютеру сигнал о нажатии клавиши.

Метод Keyboard.print() должен использоваться только после вызова Keyboard.begin().

ВНИМАНИЕ: при использовании команды Keyboard.print() Ардуино берет управление вашей клавиатурой на себя! Поэтому, перед использованием этой команды убедитесь, что у вас есть возможность отключить Ардуино от управления клавиатурой. С этой целью рекомендуется использовать какую-либо кнопку, позволяющую включать или выключать данную функцию.

Синтаксис

```
Keyboard.print(character)
Keyboard.print(characters)
```

Параметры

character: значение типа char или int, которое будет отправлено компьютеру в качестве сигнала о нажатии клавиши

characters: строка, которая будет отправлена компьютеру в виде последовательности нажатий соответствующих клавиш

Возвращаемые значения

int: количество отправленных байт

Пример

```
void setup() {
  // переводим вывод 2 в режим входа и включаем
  // подтягивающий резистор, в результате чего на выводе
присутствует высокий уровень
  // до тех пор, пока он не будет соединен с землей:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //если кнопка нажата
```

```
  if(digitalRead(2)==LOW){
    //отправляем сообщение
    Keyboard.print("Hello!");
  }
}
```

Смотрите также

Keyboard.begin()
Keyboard.end()
Keyboard.press()
Keyboard.println()
Keyboard.release()
Keyboard.releaseAll()
Keyboard.write()

Keyboard.println()

Описание

Отправляет подсоединенному компьютеру сигнал о нажатии клавиш, заканчивающийся новой строкой и символом перевода каретки.

Метод Keyboard.println() должен использоваться только после вызова Keyboard.begin().

ВНИМАНИЕ: при использовании команды Keyboard.println() Ардуино берет управление вашей клавиатурой на себя! Поэтому, перед использованием этой команды убедитесь, что у вас есть возможность отключить Ардуино от управления клавиатурой. С этой целью рекомендуется использовать какую-либо кнопку, позволяющую включать или выключать данную функцию.

Синтаксис

```
Keyboard.println()
Keyboard.println(character)
Keyboard.println(characters)
```

Параметры

character: значение типа char или int, которое будет отправлено компьютеру в качестве сигнала о нажатии клавиши, завершающегося новой строкой и символом возврата каретки.

characters: строка, которая будет отправлена компьютеру в виде последовательности нажатий соответствующих клавиш и завершающаяся новой строкой с символом возврата каретки.

Возвращаемые значения

int: количество отправленных байт

Пример

```
void setup() {
  // переводим вывод 2 в режим входа и включаем
  // подтягивающий резистор, в результате чего на выводе
присутствует высокий уровень
  // до тех пор, пока он не будет соединен с землей:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
```

```
}

void loop() {
  //если кнопка нажата
  if(digitalRead(2)==LOW){
    //отправляем сообщение
    Keyboard.println("Hello!");
  }
}
```

Смотрите также

Keyboard.begin()
Keyboard.end()
Keyboard.press()
Keyboard.print()
Keyboard.release()
Keyboard.releaseAll()
Keyboard.write()

Keyboard.release()

Описание

Эмулирует отпускание определенной клавиши. Для получения дополнительной информации см. Keyboard.press().

Синтаксис

```
Keyboard.release(key)
```

Параметры

key: клавиша, сигнал об отпускании которой необходимо отправить компьютеру.

Возвращаемые значения

int: количество "отпущенных" клавиш

Пример

```
// для OSX используйте этот параметр:
char ctrlKey = KEY_LEFT_GUI;
// для Windows и Linux используйте этот параметр:
//  char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // переводим вывод 2 в режим входа и включаем
  // подтягивающий резистор, в результате чего на выводе
присутствует высокий уровень
  // до тех пор, пока он не будет соединен с землей:
  pinMode(2, INPUT_PULLUP);
  // инициализируем контроль над клавиатурой:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
    // ничего не делаем до тех пор, пока на выводе 2 не появится
низкий уровень
    delay(500);
  }
  delay(1000);
```

```
  // новый документ:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.releaseAll();
  // ждем открытия нового окна:
  delay(1000);
}
```

Смотрите также

Keyboard.begin()
Keyboard.end()
Keyboard.press()
Keyboard.print()
Keyboard.println()
Keyboard.releaseAll()
Keyboard.write()
Клавиши-модификаторы

Keyboard.releaseAll()

Описание

Эмулирует отпускание всех нажатых клавиш. Для получения дополнительной информации см. Keyboard.press().

Синтаксис

```
Keyboard.releaseAll()
```

Параметры

нет

Возвращаемые значения

int: количество "отпущенных" клавиш

Пример

```
// для OSX используйте этот параметр:
char ctrlKey = KEY_LEFT_GUI;
// для Windows и Linux используйте этот параметр:
//  char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // переводим вывод 2 в режим входа и включаем
  // подтягивающий резистор, в результате чего на выводе
присутствует высокий уровень
  // до тех пор, пока он не будет соединен с землей:
  pinMode(2, INPUT_PULLUP);
  // инициализируем контроль над клавиатурой:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
    // ничего не делаем до тех пор, пока на выводе 2 не появится
низкий уровень
    delay(500);
  }
  delay(1000);
  // новый документ:
```

```
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.releaseAll();
  // ждем открытия нового окна:
  delay(1000);
}
```

Смотрите также

Keyboard.begin()
Keyboard.end()
Keyboard.press()
Keyboard.print()
Keyboard.println()
Keyboard.release()
Keyboard.write()
Клавиши-модификаторы

Keyboard.write()

Описание

Отправляет подсоединенному компьютеру сигнал о нажатии клавиши. В данном случае под нажатием подразумевается кратковременное нажатие клавиши на клавиатуре. Данная команда позволяет отправлять некоторые ASCII-символы, а также сигналы о нажатии специальных клавиш-модификаторов.

Метод поддерживает отправку только тех ASCII-символов, которые присутствуют на клавиатуре. Например, ASCII-код 8 (Backspace) отправится корректно, а ASCII-код 25 (замена) - нет. При отправке прописных букв, команда Keyboard.write() помимо желаемого символа отправляет сигнал о нажатии Shift, подобно набору на клавиатуре. При отправке числового значения осуществляется отправка соответствующего ASCII-символа (например, при выполнении Keyboard.write(97) произойдет отправка символа 'a').

Полный список ASCII-символов см. на странице ASCIITable.com.

ВНИМАНИЕ: при использовании команды Keyboard.write() Ардуино берет управление вашей клавиатурой на себя! Поэтому, перед использованием этой команды убедитесь, что у вас есть возможность отключить Ардуино от управления клавиатурой. С этой целью рекомендуется использовать какую-либо кнопку, позволяющую включать или выключать данную функцию.

Синтаксис

```
Keyboard.write(character)
```

Параметры

character: символ или число int, которое необходимо отправить компьютеру. Может быть представлено в любом виде, приемлемом для типа char. Все из представленных ниже примеров корректны и отправляют одно и то же значение - 65 или ASCII-символ A:

```
Keyboard.write(65);          // отправляет ASCII-код 65, или A
Keyboard.write('A');             // то же значение, но в кавычках
Keyboard.write(0x41);        // то же значение в
шестнадцатеричном виде
Keyboard.write(0b01000001); // то же значение в двоичном виде
(не самый удобный вариант, но он работает)
```

Возвращаемые значения

int: количество отправленных байт

Пример

```
void setup() {
  // переводим вывод 2 в режим входа и включаем
  // подтягивающий резистор, в результате чего на выводе
присутствует высокий уровень
  // до тех пор, пока он не будет соединен с землей:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //если кнопка нажата
  if(digitalRead(2)==LOW){
    //отправляем ASCII-символ 'A',
    Keyboard.write(65);
  }
}
```

Смотрите также

Keyboard.begin()
Keyboard.end()
Keyboard.press()
Keyboard.print()
Keyboard.println()
Keyboard.release()
Keyboard.releaseAll()
Инструкция: Таблица ASCII-символов

Библиотеки

Как и на многих других платформах, возможности среды программирования Arduino могут быть существенно расширены за счет использования библиотек. Библиотеки расширяют функциональность программ и несут в себе дополнительные функции, например, для работы с аппаратными средствами, функции по обработке данных и т.д. Ряд библиотек устанавливается автоматически вместе со средой разработки, однако вы также можете скачивать или создавать собственные библиотеки. Инструкции по установке библиотек см. здесь. См. также инструкции по написанию собственных библиотек.

Для подключения библиотеки к программе, выберите ее из меню **Sketch > Import Library**.

Стандартные библиотеки

EEPROM - чтение и запись в "постоянную" память.
Ethernet - для подсоединения к Интернету через плату расширения Arduino Ethernet.
Firmata - для взаимодействия с приложениями на компьютере по стандартному последовательному протоколу.
GSM - для соединения с сетью GSM/GRPS через GSM-плату расширения.
LiquidCrystal - для работы с жидкокристаллическими дисплеями (LCD).
SD - для чтения и записи данных на SD-карту памяти.
Servo - для управления серводвигателями.
SPI - для взаимодействия с периферийными устройствами по последовательному интерфейсу SPI.
SoftwareSerial - для реализации последовательных интерфейсов на любых цифровых выводах. Начиная с версии Ардуино 1.0, в качестве библиотеки SoftwareSerial используется библиотека NewSoftSerial (автор Mikal Hart).
Stepper - для управления шаговыми двигателями.
TFT - для вывода текста, изображений и графических примитивов на TFT-экране Arduino.
WiFi - для соединения с Интернетом через плату расширения Arduino WiFi.
Wire - библиотека для работы с двухпроводным интерфейсом (TWI/I2C), позволяющим принимать или отправлять данные между сетью устройств или датчиков.

Библиотеки Matrix и Sprite больше не входят в состав стандартного распространяемого ПО.

Специализированные библиотеки Arduino Due

Audio - проигрывание аудио-файлов с SD-карты памяти.

Scheduler - реализация многозадачности.
USBHost - взаимодействие с USB-гаджетами, такими как мышь или клавиатура.

Специализированные библиотеки Esplora

Esplora - данная библиотека позволяет легко взаимодействовать с различными датчиками и приводами на плате Arduino Esplora.

Специализированные библиотеки Arduino Robot

Robot - библиотека обеспечивает доступ к функциям Arduino Robot.

Библиотеки для работы с USB (для Leonardo, Micro, Due и Esplora)

Keyboard - отправка сигналов нажатия клавиш подсоединенному компьютеру.
Mouse - управление указателем мыши на подсоединенном компьютере.

Вспомогательные библиотеки

Для использования какой-либо из этих библиотек, необходимо сначала ее установить. Подробности процесса установки описаны в соответствующих инструкциях. См. также инструкции по написанию собственных библиотек.

Связь (сети и протоколы):

Messenger - для обработки текстовых сообщений, поступающих от компьютера.
NewSoftSerial - усовершенствованная версия библиотеки SoftwareSerial.
OneWire - управление устройствами (от Dallas Semiconductor), работающими по протоколу One Wire.
PS2Keyboard - считывание символов с PS2-клавиатуры.
Simple Message System - отправка сообщений между компьютером и Ардуино.
SSerial2Mobile - отправка текстовых сообщений и электронной почты с мобильного телефона (посредством AT-команд и библиотеки SoftwareSerial).
Webduino - реализация расширяемого веб-сервера (для использования с платой расширения Arduino Ethernet).
X10 - отправка сигналов через линии электропередач по протоколу X10.
XBee - для связи с беспроводными модулями XBees в режиме API.
SerialControl - удаленное управление другими Ардиуно по последовательному интерфейсу.

Обработка сигнала с датчиков:

Capacitive Sensing - использование двух или более выводов Ардуино в качестве емкостных датчиков.
Debounce - для считывания зашумленного сигнала с цифровых выводов (может использоваться, например, для обработки дребезга контактов при нажатии кнопки).

Дисплеи и светодиоды:

GFX - базовый класс со стандартными графическими процедурами (от Adafruit Industries).
GLCD - графические процедуры для LCD-дисплеев на основе чипсета KS0108 или эквивалентного.
Усовершенствованная библиотека LCD - исправлены ошибки инициализации LCD в официальной библиотеке LCD от Arduino.
LedControl - для управления светодиодными матрицами или семисегментными индикаторами, работающих с драйвером MAX7221 или MAX7219.
LedControl - альтернатива библиотеке Matrix для управления несколькими светодиодами с помощью микросхем Maxim.
LedDisplay - управление светодиодной бегущей строкой HCMS-29xx.
Matrix - базовая библиотека для работы с матрицей светодиодов.
PCD8544 - библиотека для работы с LCD-контроллером экранов, подобных Nokia 55100 (от Adafruit Industries).
Sprite - базовая библиотека для работы со спрайтами и анимацией на светодиодных матрицах.
ST7735 - библиотека для работы с LCD-контроллером TFT-экранов диагонально 1.8" и разрешением 128x160 (от Adafruit Industries).

Синусоидальные и аудио-сигналы:

FFT - частотный анализ аудио- и других аналоговых сигналов.
Tone - генерирование прямоугольного сигнала звуковой частоты на любом выводе микроконтроллера в фоновом режиме.

Двигатели и ШИМ:

TLC5940 - 16-канальный 12-разрядный ШИМ-контроллер.

Работа с временными интервалами:

DateTime - библиотека для отслеживания в программе текущей даты и времени.
Metro - выполнение определенных действий через равные промежутки времени.
MsTimer2 - использует прерывание от Таймера 2 для выполнения определенного действия каждые N миллисекунд.

Вспомогательные библиотеки:

PString - небольшой класс для осуществления вывода в буферы.
Streaming - метод упрощения работы с операторами вывода.

Создание собственной библиотеки для Ардуино

Этот документ поможет разобраться, как создавать библиотеки для Ардуино. Сначала будет рассмотрена программа, генерирующая сигналы азбуки Морзе, а затем даны пояснения, как вынести ее функции в отдельную библиотеку. Использование библиотек позволяет другим людям использовать написанный вами код, а также легко обновлять его по мере выхода новых версий вашей библиотеки.

Начнем с программы, генерирующий простой сигнал азбуки Морзе:

```
int pin = 13;

void setup()
{
  pinMode(pin, OUTPUT);
}

void loop()
{
  dot(); dot(); dot();
  dash(); dash(); dash();
  dot(); dot(); dot();
  delay(3000);
}

void dot()
{
  digitalWrite(pin, HIGH);
  delay(250);
  digitalWrite(pin, LOW);
  delay(250);
}

void dash()
{
  digitalWrite(pin, HIGH);
  delay(1000);
  digitalWrite(pin, LOW);
  delay(250);
}
```

Если запустить эту программу, то можно убедиться, что она подает сигнал SOS (сигнал бедствия) светодиодом, подключенным к 13 выводу.

В программе есть несколько участков, которые нам необходимо объединить в

библиотеку. Во-первых, конечно же, это функции **dot()** и **dash()**, которые и формируют сигнал. Во-вторых, это переменная **pin**, которая используется функциями для того, чтобы знать, с каким именно выводом необходимо работать. И, наконец, в программе есть вызов функции **pinMode ()**, которая заставляет работать указанный вывод в качестве выхода.

Пора бы сделать из нашей программы библиотеку!

Для этого вам понадобится, по меньшей мере, два файла: заголовочный файл (с расширением .h) и файл с исходным кодом (с расширением .cpp). Заголовочный файл представляет собой описание библиотеки: чаще всего, это просто список всего, что в ней есть. Файл-исходник содержит непосредственно программный код библиотеки. Назовем нашу библиотеку "Morse", соответственно, наш заголовочный файл будет "Morse.h". Давайте посмотрим, что внутри этого файла. Поначалу содержимое файла может показаться вам немного странным, однако все станет на свои места, как только вы увидите исходник, идущий "в комплекте".

Структура заголовочного файла представляет собой набор строк, каждая из которых соответствует одной функции библиотеки. Эти строки находятся внутри класса, который также может включать в себя все необходимые переменные:

```
class Morse
{
  public:
    Morse(int pin);
    void dot();
    void dash();
  private:
    int _pin;
};
```

Класс - это просто набор функций и переменных, собранных в одном месте. Эти функции и переменные могут быть общедоступными (*public*) - т.е. могут вызываться людьми, работающими с вашей библиотекой, или внутренними (*private*) - которые видны только в пределах самого класса. У каждого класса есть специальная функция, называемая конструктором (*constructor*), которая предназначена для создания экземпляра (*instance*) класса. Конструктор имеет такое же имя, как и класс, и не возвращает никаких значений.

Помимо этого, есть еще несколько деталей, которые необходимо включить в заголовочный файл. Одна из них - это оператор #include, который позволяет подключить к нашей библиотеке стандартные типы и константы языка Ардуино (такой оператор добавляется автоматически в коде обычных программ, но в библиотеке его нужно дописать самому). Он выглядит

примерно так (и располагается перед объявлением показанного выше класса):

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

И последнее: общепринято заключать все содержимое заголовочного файла в странную конструкцию:

```
#ifndef Morse_h
#define Morse_h

// здесь располагается оператор #include и весь остальной код...

#endif
```

По сути, это предотвращает возможные проблемы на случай, если кто-то подключит вашу библиотеку дважды.

В завершение, наверху библиотеки обычно располагают комментарий, содержащий название библиотеки, краткое описание того, что она делает, автора, дату создания и тип лицензии.

Давайте взглянем на заголовочный файл и посмотрим, что у нас получилось:

```
/*
  Morse.h - Library for flashing Morse code.
  Created by David A. Mellis, November 2, 2007.
  Released into the public domain.
*/
#ifndef Morse_h
#define Morse_h

#include "Arduino.h"

class Morse
{
  public:
    Morse(int pin);
    void dot();
    void dash();
  private:
    int _pin;
};

#endif
```

А теперь давайте разберем содержимое исходного файла - Morse.cpp.

Вначале файла идет несколько операторов #include, которые предоставляют

остальной программе доступ к стандартным функциям Ардуино и к объявлениям функций внутри заголовочного файла:

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

Затем идет конструктор, который описывает, что должно произойти, когда кто-то создаст экземпляр вашего класса. В данном случае пользователь указывает, какой именно вывод он хотел бы использовать. Мы конфигурируем этот вывод в качестве выхода и сохраняем его во внутреннюю переменную для последующего использования в других функциях:

```
Morse::Morse(int pin)
{
  pinMode(pin, OUTPUT);
  _pin = pin;
}
```

В этом коде есть непонятные моменты. Во-первых, конструкция **Morse::** перед именем функции. Это значит, что функция является частью класса **Morse**. То же самое вы увидите при объявлении других функций этого класса. Второй момент - это знак подчеркивания в имени нашей внутренней *private*-переменной, **_pin**. Вообще-то эта переменная может иметь любой имя, главное, чтобы она соответствовала имени, объявленному в заголовочном файле. Добавление подчеркивания перед именем переменной - это общепринятая методика, применяющаяся для того, чтобы явно отличать *private*-переменные. Кроме того, подчеркивание позволяет программе отличить *private*-переменную от аргумента функции (**pin** в данном случае).

Далее идет сам код из первоначальной программы (ну наконец-то!). Он выглядит абсолютно точно так же, за исключением приставки **Morse::** перед именами функций и переменной **_pin** вместо **pin**:

```
void Morse::dot()
{
  digitalWrite(_pin, HIGH);
  delay(250);
  digitalWrite(_pin, LOW);
  delay(250);
}

void Morse::dash()
{
  digitalWrite(_pin, HIGH);
  delay(1000);
  digitalWrite(_pin, LOW);
  delay(250);
```

```
}
```

В завершение, хорошим тоном считается добавление комментария в начале исходного файла. Посмотрим что получилось:

```
/*
  Morse.cpp - Library for flashing Morse code.
  Created by David A. Mellis, November 2, 2007.
  Released into the public domain.
*/

#include "Arduino.h"
#include "Morse.h"

Morse::Morse(int pin)
{
  pinMode(pin, OUTPUT);
  _pin = pin;
}

void Morse::dot()
{
  digitalWrite(_pin, HIGH);
  delay(250);
  digitalWrite(_pin, LOW);
  delay(250);
}

void Morse::dash()
{
  digitalWrite(_pin, HIGH);
  delay(1000);
  digitalWrite(_pin, LOW);
  delay(250);
}
```

И это все, что необходимо сделать (есть еще некоторые возможности, но мы поговорим о них чуть позже). Теперь попробуем нашу библиотеку в действии.

Прежде всего, создайте папку **Morse** в директории **libraries** внутри вашей рабочей папки с проектами. Скопируйте или переместите файлы Morse.h и Morse.cpp в созданную папку. Теперь запустите среду разработки Ардуино - в меню **Sketch > Import Library** вы должны увидеть библиотеку Morse. Она будет автоматически компилироваться вместе с использующими ее программами. Если этого не произойдет - проверьте ее расширение и убедитесь, что файл действительно имеет формат .cpp или .h (без дополнительных расширений вроде .pde или .txt, например).

Попробуем переписать нашу старую программу "SOS" с использованием новой библиотеки:

```
#include <Morse.h>

Morse morse(13);

void setup()
{
}

void loop()
{
  morse.dot(); morse.dot(); morse.dot();
  morse.dash(); morse.dash(); morse.dash();
  morse.dot(); morse.dot(); morse.dot();
  delay(3000);
}
```

По сравнению с предыдущей версией в программе появилось несколько отличий (кроме того факта, что часть кода перенесена в библиотеку).

Во-первых, мы добавили оператор #include в начало программы, который включает библиотеку Morse в отправляемый плате код. Поэтому, если в программе библиотека больше не используется, желательно удалить #include для экономии памяти микроконтроллера.

Во-вторых, теперь мы создаем экземпляр класса Morse с именем **morse**:

```
Morse morse(13);
```

При выполнении этой строки (а фактически, это произойдет даже до функции **setup()**) будет вызван конструктор класса Morse, которому будет передан указанный здесь аргумент (в данном случае **13**).

Обратите внимание, что теперь наша функция **setup()** пуста, поскольку вызов **pinMode()** в данном случае происходит внутри библиотеки (при создании экземпляра класса).

Ну и наконец, вызов функций **dot()** и **dash()** теперь необходимо предварять префиксом **morse.** - именем того экземпляра, который мы хотим использовать. Мы можем создать несколько экземпляров класса Morse, каждый со своим выводом, хранимым во внутренней переменной _pin только в пределах этого экземпляра. Указывая определенный экземпляр класса при вызове функции, мы тем самым задаем, переменными какого экземпляра должна оперировать та или иная функция. То есть, если у нас два экземпляра:

```
Morse morse(13);
Morse morse2(12);
```

то внутри функции **morse2.dot()** переменная **_pin** будет равна 12.

При написании новой программы, вы, наверняка заметите, что среда разработки не распознает и не подсвечивает элементы созданной нами библиотеки. К сожалению, IDE Ардуино не умеет автоматически распознавать и интерпретировать то, что мы объявили внутри библиотеки (кстати, было бы хорошо добавить эту функцию), поэтому ей нужно немного помочь. Для этого создайте файл **keywords.txt** в директории Morse и запишите в него следующее:

```
Morse   KEYWORD1
dash    KEYWORD2
dot     KEYWORD2
```

Каждая строка должна содержать ключевое слово, символ табуляции (не пробелы) и тип ключевого слова. Классы подсвечиваются оранжевым и должны иметь тип KEYWORD1; функции - коричневым и должны быть типа KEYWORD2. Для того, чтобы внесенные изменения вступили в силу, необходимо перезапустить среду Ардуино.

Также неплохо было бы снабдить библиотеку примером работы с ней. Для этого, создайте папку **examples** в директории **Morse** и переместите (либо скопируйте) в нее папку с нашей программой (назовем ее **SOS**). (Отыскать программу можно с помощью команды **Sketch > Show Sketch Folder**). Если вы перезапустите среду Ардуино (честно слово, это в последний раз) - то увидите пункт **Library-Morse** в меню **File > Sketchbook > Examples** с вашим примером. Можете добавить немного комментариев, объясняющих, как пользоваться вашей библиотекой.

Если вы захотите посмотреть готовую библиотеку (с примером и ключевыми словами) - можно сказать ее отсюда: Morse.zip.

На сегодня, пожалуй, это все, но в ближайшее время возможно появится расширенное руководство по созданию библиотек. А тем временем, если у вас возникнут проблемы или появятся предложения, пожалуйста, пишите их на форум.

pinMode()

Описание

Конфигурирует режим работы указанного вывода: как вход либо как выход. Подробнее о функциональности выводов смотрите описание цифровых выводов.

В Ардуино версии 1.0.1 есть возможность задействовать внутренние подтягивающие резисторы с помощью режима INPUT_PULLUP. Соответственно, режим INPUT просто отключает внутреннюю подтяжку.

Синтаксис

```
pinMode(pin, mode)
```

Параметры

pin: номер вывода, режим работы которого будет конфигурироваться.

mode: принимает значения INPUT, OUTPUT или INPUT_PULLUP (см. страницу цифровые выводы для подробного описания их функциональности).

Возвращаемые значения

Нет

Пример

```
int ledPin = 13;                  // Светодиод подсоединен к
цифровому выводу 13

void setup()
{
  pinMode(ledPin, OUTPUT);       // устанавливаем режим работы
вывода, как "выход"
}

void loop()
{
  digitalWrite(ledPin, HIGH);    // включаем светодиод
  delay(1000);                   // ждем 1 секунду
  digitalWrite(ledPin, LOW);     // выключаем светодиод
  delay(1000);                   // ждем 1 секунду
```

```
}
```

Примечание

Выводы, являющиеся аналоговыми входами, могут также использоваться как цифровые выводы под именем A0, A1 и т.д.

Смотрите также

константы
digitalWrite()
digitalRead()
Инструкция: Описание выводов платы Arduino

digitalWrite()

Описание

Отправляет на цифровой вывод значение HIGH или LOW.

Если функцией pinMode() вывод сконфигурирован как выход (OUTPUT), то при выполнении функции digitalWrite() его напряжение будет изменено на соответствующее значение: 5 В (либо 3.3 В для плат, работающих от 3.3В) при отправке HIGH, 0 В (земля) - при LOW.

Если вывод сконфигурирован как вход INPUT, то отправка функцией digitalWrite () значения HIGH приведет к подключению внутреннего подтягивающего резистора номиналом 20 КОм (см. инструкцию по цифровым выводам). Запись значения LOW приведет к отключению подтяжки. Внутренний подтягивающий резистор может обеспечить только тусклое свечение светодиода. Поэтому, если светодиод горит, но очень тускло, наиболее вероятная причина этого - подтягивающий резистор. Для решения данной проблемы необходимо перевести соответствующий вывод в режим выхода с помощью функции pinMode().

ПРИМЕЧАНИЕ: Существуют некоторые сложности при использовании вывода 13 в качестве цифрового входа. Причиной этого является светодиод и резистор, которые припаяны к этому выводу на большинстве плат Ардуино. При включении внутреннего подтягивающего резистора 20 КОм, напряжение на этом выводе установится на уровне около 1.7 В, вместо ожидаемых 5 В, поскольку светодиод и последовательно соединенный резистор на плате понижают уровень напряжения. Таким образом, вывод будет всегда находится в состоянии LOW. Поэтому, чтобы использовать вывод 13 в качестве цифрового входа, необходимо использовать внешний резистор на землю.

Синтаксис

```
digitalWrite(pin, value)
```

Параметры

pin: номер вывода

value: значение HIGH или LOW

Возвращаемые значения

нет

Пример

```
int ledPin = 13;                  // светодиод подключен к выводу
13

void setup()
{
  pinMode(ledPin, OUTPUT);        // переключаем цифровой вывод в
режим выхода
}

void loop()
{
  digitalWrite(ledPin, HIGH);     // включаем светодиод
  delay(1000);                    // ждем 1 секунду
  digitalWrite(ledPin, LOW);      // выключаем светодиод
  delay(1000);                    // ждем 1 секунду
}
```

Программа устанавливает на выводе 13 высокий уровень HIGH, выдерживает паузу в 1 секунду, после чего возвращает вывод в низкий уровень LOW.

Примечание

Выводы, являющиеся аналоговыми входами, могут также использоваться как цифровые выводы под именем A0, A1 и т.д.

Смотрите также

pinMode()
digitalRead()
Инструкция: Цифровые выводы

digitalRead()

Описание

Считывает уровень сигнала HIGH или LOW с указанного цифрового вывода.

Синтаксис

```
digitalRead(pin)
```

Параметры

pin: номер цифрового вывода, с которого необходимо считать значение (*int*)

Возвращаемые значения

HIGH или LOW

Пример

Программа устанавливает на выводе 13 тот же уровень сигнала, что и на выводе 7.

```
int ledPin = 13; // светодиод подсоединен к цифровому выводу 13
int inPin = 7;   // кнопка подсоединена к цифровому выводу 7
int val = 0;     // переменная для хранения считанного значения

void setup()
{
  pinMode(ledPin, OUTPUT);      // конфигурируем цифровой вывод
13 как выход
  pinMode(inPin, INPUT);      // конфигурируем цифровой вывод 13
как вход
}

void loop()
{
  val = digitalRead(inPin);   // считываем значение со входа
  digitalWrite(ledPin, val);    // выводим на светодиод уровень
сигнала на кнопке
}
```

Примечание

Если вывод ни к чему не присоединен, функция digitalRead() может вернуть случайную величину, как HIGH, так и LOW.

Выводы, являющиеся аналоговыми входами, могут также использоваться как цифровые выводы под именем A0, A1 и т.д.

Смотрите также

pinMode()
digitalWrite()
Инструкция: Цифровые выводы

analogReference(type)

Описание

Устанавливает источник опорного напряжения, использующийся при считывании аналогового сигнала (другими словами, задает максимальное значение входного диапазона). Для выбора источника опорного напряжения доступны следующие значения:

DEFAULT: опорное напряжение по умолчанию, равное 5 В (на 5В-платах Ардуино) или 3.3 В (на 3.3В-платах Ардуино)
INTERNAL: внутренне опорное напряжение, равное 1.1 В в микроконтроллерах ATmega168 и ATmega328, или 2.56 В в микроконтроллере ATmega8 (*не доступно в Arduino Mega*)
INTERNAL1V1: внутреннее опорное напряжение 1.1 В (*только для Arduino Mega*)
INTERNAL2V56: внутреннее опорное напряжение 2.56 В (*только для Arduino Mega*)
EXTERNAL: в качестве опорного напряжения будет использоваться напряжение, приложенное к выводу AREF (от 0 до 5В)

Параметры

type: тип источника опорного напряжения (DEFAULT, INTERNAL, INTERNAL1V1, INTERNAL2V56 или EXTERNAL).

Возвращаемые значения

Нет.

Примечание

Сразу после изменения источника опорного напряжения, несколько первых значений, возвращаемых функцией analogRead(), могут быть неточными.

Предупреждение

При использовании внешнего источника опорного напряжения, напряжение на выводе AREF должно быть строго в пределах от 0 до 5 В! При этом перед вызовом функции analogRead() нужно обязательно установить тип источника как EXTERNAL. В противном случае, возможно короткое замыкание внутреннего источника опорного напряжения с выводом AREF, что может привести к повреждению микроконтроллера на вашей плате

Ардуино.

Подобную ситуацию также можно предотвратить, если внешний источник опорного напряжения соединять с выводом AREF через резистор номиналом 5 кОм. Такое соединение даст возможность переключаться между внутренним и внешним опорным напряжением. Однако, при этом следует иметь ввиду, что резистор изменит величину опорного напряжения, поскольку вывод AREF соединяется с внутренним резистором номиналом 32 КОм. Два резистора образуют делитель напряжения, таким образом, например, 2.5В, приложенные через резистор, в итоге дадут 2.5 * 32 / (32 + 5) = ~2,2 В на выводе AREF.

Смотрите также

Описание аналоговых входов
analogRead()

analogRead()

Описание

Считывает величину напряжения с указанного аналогового вывода. В составе Ардуино есть 6-канальный (8-канальный - в Mini и Nano, 16 - в Mega) 10-битный аналогово-цифровой преобразователь, который преобразовывает входное напряжение из диапазона 0 - 5 В в целочисленные значения в пределах от 0 до 1023 соответственно. Разрешающая способность АЦП составляет: 5 В / 1024 значения или 0.0049 В (4.9 мВ) на одно значение. Входной диапазон и разрешающая способность могут меняться с помощью функции analogReference().

Для считывания значения с аналогового входа требуется около 100 микросекунд (0.0001 с), поэтому максимальная частота опроса вывода приблизительно равна 10 000 раз в секунду.

Синтаксис

```
analogRead(pin)
```

Параметры

pin: номер вывода, с которого будет считываться напряжение (0 - 5 для большинства плат, 0 - 7 для Mini и Nano, 0 - 15 для Mega)

Возвращаемые значения

целое число int (от 0 до 1023)

Примечание

Если аналоговый вход ни к чему не подключен, значение, возвращаемое функцией analogRead(), будет меняться под влиянием нескольких факторов (таких, как величина напряжения на других аналоговых входах, наводок от вашей руки вблизи платы и т.д.).

Пример

```
int analogPin = 3;     // ползунок потенциометра (средний вывод) подключен к аналоговому выводу 3
                       // крайние выводы соединены с землей и +5В
```

```
int val = 0;            // переменная для хранения считанного
значения

void setup()
{
  Serial.begin(9600);          //  настройка последовательного
соединения
}

void loop()
{
  val = analogRead(analogPin);    // считываем напряжение с
аналогового входа
  Serial.println(val);            // наблюдаем считанное
значение
}
```

Смотрите также

analogReference()
Инструкция: Аналоговые входы

analogWrite()

Описание

Формирует заданное аналоговое напряжение на выводе в виде ШИМ-сигнала. Может использоваться для варьирования яркости свечения светодиода или управления скоростью вращения двигателя. После вызова analogWrite(), на выводе будет непрерывно генерироваться ШИМ-сигнал с заданным коэффициентом заполнения до следующего вызова функции analogWrite() (либо до момента вызова digitalRead() или digitalWrite(), взаимодействующих с этим же выводом). Частота ШИМ составляет приблизительно 490 Гц.

На большинстве плат Arduino (на базе микроконтроллеров ATmega168 или ATmega328) функция analogWrite() работает с выводами 3, 5, 6, 9, 10 и 11. На Arduino Mega функция работает с выводами со 2 по 13. На более старых версиях Arduino (на базе микроконтроллера ATmega8) функция analogWrite() работает только с выводами 9, 10 и 11.

Arduino Due поддерживает функцию analogWrite() для выводов со 2 по 13, а также для выводов DAC0 и DAC1. В отличие от ШИМ-выводов, DAC0 и DAC1 являются выводам цифро-аналоговых преобразователей, поэтому при вызове analogWrite() ведут себя как обычные аналоговые выходы.

При работе с analogWrite() предварительный вызов функции pinMode() для переключения выводов в режим «выход» не требуется.

Функция analogWrite() не имеет ничего общего с аналоговыми выводами и функцией analogRead().

Синтаксис

```
analogWrite(pin, value)
```

Параметры

pin: вывод, на котором будет формироваться напряжение.

value: коэффициент заполнения – лежит в пределах от 0 (всегда выключен) до 255 (всегда включен).

Возвращаемые значения

нет

Примечания и известные проблемы

На выводах 5 и 6 генерируется ШИМ-сигнал с коэффициентом заполнения большим, чем заданное ожидаемое значение. Это происходит в результате взаимодействия с функциями millis() и delay(), которые используют тот же внутренний таймер, что применяется для генерирования ШИМ-сигнала. Данный эффект более ярко выражен при малых значениях задаваемого коэффициента заполнения (0 - 10) и может проявляться в неполном выключении выводов 5 и 6 при коэффициенте равном 0.

Пример

Формирование на выводе, управляющим светодиодом, напряжения пропорционального напряжению на потенциометре.

```
int ledPin = 9;       // светодиод подключен к цифровому выводу 9
int analogPin = 3;    // потенциометр подключен к аналоговому
выводу 3
int val = 0;          // переменная для хранения считанного
значения

void setup()
{
  pinMode(ledPin, OUTPUT);   // переключение вывода в режим
«выход»
}

void loop()
{
  val = analogRead(analogPin);   // считываем входное напряжение
  analogWrite(ledPin, val / 4);  /* значения, возвращаемые
analogRead лежат в пределах от 0 до 1023,
                        а задаваемый коэффициент analogWrite -
от 0 to 255*/
}
```

Смотрите также

analogRead()
Инструкция: ШИМ

analogReadResolution()

Описание

analogReadResolution() - это функция, расширяющая API для работы с аналоговыми величинами для Arduino Due.

Функция устанавливает размерность (в битах) значения, возвращаемого функцией analogRead(). Для обратной совместимости с платами на базе AVR-микроконтроллеров эта размерность, по умолчанию, составляет 10 бит (диапазон чисел 0-1023).

В состав Due входит 12-битный АЦП, задействовать возможности которого можно путем изменения размерности на 12 бит. Это позволит функции analogRead() возвращать значения в диапазоне от 0 до 4095.

Синтаксис

```
analogReadResolution(bits)
```

Параметры

bits: размерность (в битах) значения, возвращаемого функцией analogRead(). Может быть в пределах от 1 до 32. Допускается задавать размерность больше 12, но в этом случае значения, возвращаемые analogRead(), будут подвергаться аппроксимации. См. примечание ниже.

Возвращаемые значения

Нет.

Примечание

Если размерность, указанная в функции analogReadResolution(), превышает возможности вашей платы, Ардуино будет возвращать результаты максимального разрешения, заполняя незначащие биты нулями.

Например: при вызове analogReadResolution(16), Arduino Due будет выдавать аппроксимированное 16-битное число, первые 12 бит которого будут содержать **фактические** показания АЦП, а последние 4 бита **будут заполнены нулями**.

Если же размерность, указанная в функции analogReadResolution(), меньше максимально возможной, то младшие биты значений, возвращаемых АЦП,

будут **отбрасываться**.

Использование размерности 16 бит (или любой другой, превышающей возможности текущего устройства) способствует легкой переносимости кода и позволяет программам автоматически поддерживать АЦП большей разрядности на других устройствах без изменения программы.

Пример

```
void setup() {
  // открываем последовательное соединение
  Serial.begin(9600);
}

void loop() {
  // считываем напряжение со входа A0, размерность по умолчанию
(10 бит)
  // и отправляем считанное значение через последовательный порт
  analogReadResolution(10);
  Serial.print("ADC 10-bit (default) : ");
  Serial.print(analogRead(A0));

  // изменяем размерность на 12 бит и считываем A0
  analogReadResolution(12);
  Serial.print(", 12-bit : ");
  Serial.print(analogRead(A0));

  // изменяем размерность на 16 бит и считываем A0
  analogReadResolution(16);
  Serial.print(", 16-bit : ");
  Serial.print(analogRead(A0));

  // изменяем размерность на 8 бит и считываем A0
  analogReadResolution(8);
  Serial.print(", 8-bit : ");
  Serial.println(analogRead(A0));

  // небольшая задержка
  delay(100);
}
```

Смотрите также

Описание аналоговых входов
analogRead()

analogWriteResolution()

Описание

analogWriteResolution() - это функция, расширяющая API для работы с аналоговыми величинами для Arduino Due.

analogWriteResolution() устанавливает размерность значений, передаваемых функции analogWrite(). С целью обратной совместимости с платами на базе AVR-микроконтроллеров эта размерность, по умолчанию, составляет 8 бит (диапазон значений 0-255) .

Due имеет следующие аппаратные возможности:

12 выводов, по умолчанию принимающие 8-битные значения для вывода ШИМ-сигнала (как и на AVR-платах). Размерность может быть изменена на 12 бит.
2 вывода 12-разрядного ЦАП (цифро-аналогового преобразователя)

Изменив размерность на 12, можно использовать функцию analogWrite() со значениями в диапазоне от 0 до 4095. Это позволит не только задействовать всю шкалу ЦАП, но и задавать более точные значения ШИМ-сигнала.

Синтаксис

```
analogWriteResolution(bits)
```

Параметры

bits: размерность (в битах) значений, передаваемых функции analogWrite(). Может быть в пределах от 1 до 32. Допускается задавать размерности, которые не соответствуют аппаратным возможностям вашей платы. В этом случае значение, передаваемое analogWrite() будет либо отсекаться (если размерность слишком велика), либо дополняться нулями (если слишком мала). См. примечание ниже.

Возвращаемые значения

Нет.

Примечание

Если размерность, указанная в функции analogWriteResolution(), превышает возможности вашей платы, Ардуино просто **отбросит** лишние биты.

Например: после вызова analogWriteResolution(16) на Arduino Due, при отправке 16-битного значения 12-битному ЦАП функцией analogWrite() будут использованы только первые 12 бит, остальные же 4 бита будут отброшены.

Если же размерность, указанная в функции analogWriteResolution(), меньше максимально возможной, то недостающие биты будут **дополнены** нулями. Например: после вызова analogWriteResolution(8), для отправки 8-битного значения 12-битному ЦАП Arduino Due добавит дополнительных 4 нулевых бита к значению, передаваемому функции analogWrite() .

Пример

```
void setup(){
  // открываем последовательное соединение
  Serial.begin(9600);
  // переводим цифровые выводы в режим "выход"
  pinMode(11, OUTPUT);
  pinMode(12, OUTPUT);
  pinMode(13, OUTPUT);
}

void loop(){
  // считываем значение со входа A0 и масштабируем его для ШИМ-
вывода,
  // к которому подсоединен светодиод
  int sensorVal = analogRead(A0);
  Serial.print("Analog Read) : ");
  Serial.print(sensorVal);

  // размерность ШИМ по умолчанию
  analogWriteResolution(8);
  analogWrite(11, map(sensorVal, 0, 1023, 0 ,255));
  Serial.print(" , 8-bit PWM value : ");
  Serial.print(map(sensorVal, 0, 1023, 0 ,255));

  // изменяем размерность ШИМ на 12 бит
  // полная шкала в 12 бит поддерживается только
  // Arduino Due
  analogWriteResolution(12);
  analogWrite(12, map(sensorVal, 0, 1023, 0, 4095));
  Serial.print(" , 12-bit PWM value : ");
  Serial.print(map(sensorVal, 0, 1023, 0, 4095));

  // изменяем размерность ШИМ на 4 бита
  analogWriteResolution(4);
  analogWrite(13, map(sensorVal, 0, 1023, 0, 127));
  Serial.print(", 4-bit PWM value : ");
```

```
  Serial.println(map(sensorVal, 0, 1023, 0, 127));

  delay(5);
}
```

Смотрите также

Описание аналоговых входов
analogWrite()
analogRead()
map()

tone()

Описание

Генерирует на выводе прямоугольный сигнал заданной частоты (с коэффициентом заполнения 50%). Функция также позволяет задавать длительность сигнала. Однако, если длительность сигнала не указана, он будет генерироваться до тех пор, пока не будет вызвана функция noTone(). Для воспроизведения звука вывод можно подключить к зуммеру или динамику.

В каждый момент времени может генерироваться только один сигнал заданной частоты. Если сигнал уже генерируется на каком-либо выводе, то использование функции tone() для этого вывода просто приведет к изменению частоты этого сигнала. В то же время вызов функции tone() для другого вывода не будет иметь никакого эффекта.

Использование функции tone() может влиять на ШИМ-сигнал на выводах 3 и 11 (на всех платах, кроме Mega).

ПРИМЕЧАНИЕ: для воспроизведение разных звуков на нескольких выводах, необходимо сперва вызывать noTone() на одном выводе и только после этого использовать функцию tone() на следующем.

Синтаксис

```
tone(pin, frequency)
tone(pin, frequency, duration)
```

Параметры

pin: вывод, на котором будет генерироваться сигнал

frequency: частота сигнала в Герцах - *unsigned int*

duration: длительность сигнала в миллисекундах (опционально) - *unsigned long*

Возвращаемые значения

нет

Смотрите также

noTone()
analogWrite()
Инструкция: Воспроизведение мелодии

noTone()

Описание

Прекращает генерирование прямоугольного сигнала после использования функции tone(). Если сигнал не генерируется, функция ни к чему не приводит.

ПРИМЕЧАНИЕ: для воспроизведение разных звуков на нескольких выводах, необходимо сперва вызывать noTone() на одном выводе и только после этого использовать функцию tone() на следующем.

Синтаксис

```
noTone(pin)
```

Параметры

pin: вывод, на котором следует прекратить генерирование сигнала

Возвращаемые значения

нет

Смотрите также

tone()

shiftOut()

Описание

Осуществляет побитовый сдвиг и вывод байта данных, начиная с самого старшего (левого) или младшего (правого) значащего бита. Функция поочередно отправляет каждый бит на указанный вывод данных, после чего формирует импульс (высокий уровень, затем низкий) на тактовом выводе, сообщая внешнему устройству о поступлении нового бита.

Примечание: Для взаимодействия с устройствами, тактируемыми по фронту импульсов, перед вызовом shiftOut() необходимо убедиться, что тактовый вывод переключен в низкий уровень, например с помощью функции digitalWrite(clockPin, LOW).

Функция является программной реализацией SPI; аппаратная версия реализована в библиотеке SPI, поэтому она является быстрее, но работает только со специальными выводами.

Синтаксис

```
shiftOut(dataPin, bitOrder, value)
```

Параметры

dataPin: вывод, которому будет отправляться каждый бит из сдвигаемого байта данных (*int*)

clockPin: тактовый вывод, который будет переключаться каждый раз, когда на выводе **dataPin** устанавливается корректное значение (*int*)

bitOrder: характеризует порядок, в котором будут сдвигаться и выводиться биты; может принимать значения **MSBFIRST** или **LSBFIRST**. (Most Significant Bit First - старший значащий бит первым, или Least Significant Bit - младший значащий бит первым)

value: сдвигаемый байт данных (*byte*)

Возвращаемые значения

нет

Примечание

Выводы **dataPin** и **clockPin** должны быть уже сконфигурированы как выходы с помощью функции pinMode().

На данный момент функция **shiftOut** позволяет выводить только 1 байт (8 бит), поэтому для вывода значений, больших 255, требуется два этапа:

```
// Последовательная передача в режиме MSBFIRST
int data = 500;
// сдвигаем и выводим старший байт
shiftOut(dataPin, clock, MSBFIRST, (data >> 8));
// сдвигаем и выводим младший байт
shiftOut(dataPin, clock, MSBFIRST, data);

// Последовательная передача в режиме LSBFIRST
data = 500;
// сдвигаем и выводим младший байт
shiftOut(dataPin, clock, LSBFIRST, data);
// сдвигаем и выводим старший байт
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

Пример

Схема, соответствующая примеру, описана в инструкции по работе со сдвиговым регистром 74HC595.

```
//
**********************************************************
//
//  Название : shiftOutCode, Hello
World                              //
//  Автор    : Carlyn Maw,Tom
Igoe                                    //
//  Дата     : 25 октября
2006                                       //
//  Версия   :
1.0                                                 //
//  Заметки  : Программа использования сдвигового регистра
74HC595 //
//           : для счета от 0 to
255                                  //
//
**********************************************************

//Вывод соединен с выводом ST_CP микросхемы 74HC595
int latchPin = 8;
//Вывод соединен с выводом SH_CP микросхемы 74HC595
int clockPin = 12;
////Вывод соединен с DS микросхемы 74HC595
int dataPin = 11;
```

```
void setup() {
  //переключение выводов в режим работы "вывод", т.к. к ним идет
обращение в главном цикле
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}

void loop() {
  //процедура последовательного счета
  for (int j = 0; j < 256; j++) {
    //формируем ноль на latchPin и удерживаем его до конца
передачи
    digitalWrite(latchPin, LOW);
    shiftOut(dataPin, clockPin, LSBFIRST, j);
    //возвращаем высокий уровень на latchin, тем самым сообщая
микросхеме о том, что
    //больше не требуется воспринимать информацию
    digitalWrite(latchPin, HIGH);
    delay(1000);
  }
}
```

Смотрите также

shiftIn()
SPI

shiftIn()

Описание

Осуществляет побитовый сдвиг и считывание байта данных, начиная с самого старшего (левого) или младшего (правого) значащего бита. Процесс считывания каждого бита заключается в следующем: тактовый вывод переводится в высокий уровень, считывается очередной бит из линии данных, после чего тактовый вывод сбрасывается в низкий уровень.

Примечание: функция является программной реализацией SPI; для программирования Ардуино также существует библиотека SPI, представляющую собой аппаратную реализацию, которая является быстрее, но при этом работает только со специальными выводами.

Синтаксис

```
byte incoming = shiftIn(dataPin, clockPin, bitOrder)
```

Параметры

dataPin: вывод, с которого будет считываться каждый бит (*int*)

clockPin: тактовый вывод, который будет переключаться при считывании с **dataPin**

bitOrder: порядок, в котором будут сдвигаться и считываться биты; может принимать значения **MSBFIRST** или **LSBFIRST**. (Most Significant Bit First - старший значащий бит первым, или Least Significant Bit - младший значащий бит первым)

Возвращаемые значения

считанное значение (*byte*)

Смотрите также

shiftOut()
SPI

pulseIn()

Описание

Считывает длительность импульса (любого - HIGH или LOW) на выводе. Например, если заданное значение (**value**) - **HIGH**, то функция **PulseIn()** ожидает появления на выводе сигнала **HIGH**, затем засекает время и ожидает переключения вывода в состояние **LOW**, после чего останавливает отсчет времени. Функция возвращает длительность импульса в микросекундах, либо 0 в случае отсутствия импульса в течение определенного таймаута.

Эмпирическим путем установлено, что при использовании функции для измерения широких импульсов возможно возникновение ошибок. Функция работает с импульсами длительностью от 10 микросекунд до 3 минут.

Синтаксис

```
pulseIn(pin, value)
pulseIn(pin, value, timeout)
```

Параметры

pin: номер вывода, с которого необходимо считыть импульс (*int*)

value: тип считываемого импульса: HIGH или LOW (*int*)

timeout (опционально): время ожидания импульса в микросекундах; значение по умолчанию - одна секунда (*unsigned long*)

Возвращаемые значения

длительность импульса (в микросекундах) либо 0 в случае отсутствия импульса в течение таймаута (unsigned long)

Пример

```
int pin = 7;
unsigned long duration;

void setup()
{
  pinMode(pin, INPUT);
}
```

```
void loop()
{
  duration = pulseIn(pin, HIGH);
}
```

millis()

Описание

Возвращает количество миллисекунд, прошедших с момента старта программы Ардуино. Возвращаемое число переполнится (сбросится в 0) спустя приблизительно 50 дней.

Параметры

Нет

Возвращаемые значения

Количество миллисекунд, прошедших с момента старта программы (*unsigned long*)

Пример

```
unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = millis();
  //выводим время с момента старта программы
  Serial.println(time);
  // ждем 1 секунду, чтобы не отправлять большой массив данных
  delay(1000);
}
```

Совет:

Помните, что значение, возвращаемое функцией millis(), имеет тип unsigned long. При попытке выполнения математических операций между этим значением и значениями другого типа (например, int) будет сгенерирована ошибка.

Смотрите также

micros()

delay()
delayMicroseconds()
Инструкция: Мигание без Delay

micros()

Описание

Возвращает количество микросекунд, прошедших с момента начала выполнения программы Arduino. Возвращаемое число переполнится (сбросится в 0) спустя приблизительно 70 минут. На платах Arduino с тактовой частотой 16 МГц (Duemilanove и Nano) разрешение этой функции составляет четыре микросекунды (т.е. возвращаемое значение будет всегда кратно четырем). На платах Ардуино с тактовой частотой 8 МГц (LilyPad), разрешение функции составляет восемь микросекунд.

Примечание: в одной миллисекунде 1000 микросекунд, а в одной секунде - 1 000 000 микросекунд.

Параметры

Нет

Возвращаемые значения

Количество микросекунд, прошедших с момента старта программы (*unsigned long*)

Пример

```
unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = micros();
  //выводим время с момента старта программы
  Serial.println(time);
  // ждем 1 секунду, чтобы не отправлять большой массив данных
  delay(1000);
}
```

Смотрите также

millis()
delay()

delayMicroseconds()

delay()

Описание

Приостанавливает выполнение программы на указанный промежуток времени (в миллисекундах). (В 1 секунде - 1000 миллисекунд.)

Синтаксис

```
delay(ms)
```

Параметры

ms: количество миллисекунд, на которые необходимо приостановить программу (*unsigned long*)

Возвращаемые значения

нет

Пример

```
int ledPin = 13;                  // Светодиод подсоединен к цифровому выводу 13

void setup()
{
  pinMode(ledPin, OUTPUT);       // Конфигурируем цифровой вывод как выход
}

void loop()
{
  digitalWrite(ledPin, HIGH);    // включаем светодиод
  delay(1000);                   // ждем секунду
  digitalWrite(ledPin, LOW);     // выключаем светодиод
  delay(1000);                   // ждем секунду
}
```

Предупреждение

С помощью функции delay() заставить мигать светодиод достаточно просто. Помимо этого, во многих программах функция задержки используется для таких задач, как обработка дребезга контактов и пр. Несмотря на это,

использование функции delay() в коде программы имеет существенные недостатки. В процессе действия delay() такие операции, как считывание данных с датчиков, математические вычисления или операции с выводами не могут выполняться. Фактически, функция delay() приводит к остановке практически всех операций. Альтернативный способ контролировать время - использование функции millis() (смотрите пример кода, приведенный ниже). Опытные программисты обычно избегают использования delay() для установки временных интервалов больше нескольких десятков миллисекунд (за исключением очень простых программ Arduino).

Тем не менее, некоторые события и участки кода могут работать и в процессе выполнения микроконтроллером функции delay(), т.к. эта функция не влияет на работу прерываний. Так, по прежнему будут срабатывать прерывания, записываться данные, поступающие на вывод RX по последовательному интерфейсу, а также будет поддерживаться ШИМ-сигнал, формируемый функцией analogWrite().

Смотрите также

millis()
micros()
delayMicroseconds()
Пример: Мигание без Delay

delayMicroseconds()

Описание

Приостанавливает выполнение программы на указанный промежуток времени (в микросекундах). В одной миллисекунде 1000 миллисекунд, и 1 000 000 микросекунд.

На данный момент наибольшее число, позволяющее сформировать точную задержку, - 16383. В будущих версиях Ардиуно этот показатель может быть изменен. Для создания задержек длительностью больше, чем несколько тысяч микросекунд, используйте функцию delay().

Синтаксис

```
delayMicroseconds(us)
```

Параметры

us: количество микросекунд, на которые необходимо приостановить программу (*unsigned int*)

Возвращаемые значения

нет

Пример

```
int outPin = 8;                 // цифровой вывод 8

void setup()
{
  pinMode(outPin, OUTPUT);      // конфигурируем цифровой вывод
как выход
}

void loop()
{
  digitalWrite(outPin, HIGH);   // включаем вывод
  delayMicroseconds(50);        // задержка в 50 микросекунд
  digitalWrite(outPin, LOW);    // выключаем вывод
  delayMicroseconds(50);        // задержка в 50 микросекунд
}
```

Вывод номер 8, сконфигурированный как выход, формирует

последовательность импульсов с периодом 100 микросекунд

Предупреждения и известные проблемы

Данная функция работает с высокой точностью в диапазоне от 3 микросекунд и выше. При более коротких задержках точная работа delayMicroseconds() не гарантируется.

Начиная с версии Arduino 0018, функция delayMicroseconds() больше не отключает прерывания.

Смотрите также

millis()
micros()
delay()

min(x, y)

Описание

Вычисляет минимальное значение из двух чисел.

Параметры

x: первое число, любой тип данных

y: второе число, любой тип данных

Возвращаемые значения

Меньшее из двух чисел.

Пример

```
sensVal = min(sensVal, 100); // присваивает sensVal меньшее из
чисел sensVal и 100
                             // позволяя убедиться, что значение
sensVal никогда не превысит 100
```

Примечание

Вопреки возможному интуитивному желанию, функция max() часто используется для создания нижнего предела диапазона значений переменной, а функция min() - наоборот, для создания верхнего предела.

Предупреждение

Реализация функции min() запрещает указывать другие функции в качестве параметров в скобках - это приведет к некорректным результатам:

```
min(a++, 100);   // избегайте этого - функция выдаст
некорректный результат

a++;
min(a, 100);     // вместо этого - осуществляйте математические
вычисления за пределами функции
```

Смотрите также

max()
constrain()

max(x, y)

Описание

Вычисляет максимальное значение из двух чисел.

Параметры

x: первое число, любой тип данных

y: второе число, любой тип данных

Возвращаемые значения

Большее из двух указанных чисел.

Пример

```
sensVal = max(senVal, 20); // присваивает sensVal большее из
двух чисел sensVal и 20
                           // позволяя убедиться, что значение
sensVal будет не меньше 20
```

Примечание

Вопреки возможному интуитивному желанию, функция max() часто используется для создания нижнего предела диапазона значений переменной, а функция min() - наоборот, для создания верхнего предела.

Предупреждение

Реализация функции max() запрещает указывать другие функции в качестве параметров в скобках - это приведет к некорректным результатам:

```
max(a--, 0);   // избегайте этого - функция выдаст некорректный
результат

a--;
max(a, 0);    // вместо этого - осуществляйте математические
вычисления за пределами функции
```

Смотрите также

min()
constrain()

abs(x)

Описание

Вычисляет абсолютную величину (модуль) числа.

Параметры

x: число

Возвращаемые значения

x: если **x** больше или равен 0.

-x: если **x** меньше 0.

Предупреждение

Реализация функции abs() запрещает указывать другие функции в качестве параметров в скобках - это приведет к некорректным результатам:

```
abs(a++);        // избегайте этого - функция выдаст некорректный результат

a++;
abs(a);         // вместо этого осуществляйте математические вычисления за пределами функции
```

constrain(x, a, b)

Описание

Ограничивает значение переменной заданными пределами.

Параметры

x: переменная, значение которой необходимо ограничить, любой тип данных

a: нижний предел, любой тип данных

b: верхний предел, любой тип данных

Возвращаемые значения

x: если **x** лежит в пределах между **a** и **b**.

a: если **x** меньше **a**.

b: если **x** больше **b**.

Пример

```
sensVal = constrain(sensVal, 10, 150);
// числовые показания датчика ограничены диапазоном от 10 до 150
```

Смотрите также

min()
max()

map(value, fromLow, fromHigh, toLow, toHigh)

Описание

Преобразовывает значение переменной из одного диапазона в другой. Т.е. значение переменной **value**, равное **fromLow**, будет преобразовано в число **toLow**, а значение **fromHigh** - в **toHigh**. Все промежуточные значения **value** масштабируются относительного нового диапазона [toLow; toHigh].

Функция не ограничивает значение переменной заданными пределами, поскольку ее значения вне указанного диапазона иногда несут полезную информацию. Для ограничения диапазона необходимо использовать функцию constrain() либо до, либо после функции map().

Обратите внимание, что нижние пределы указываемых диапазонов (fromLow, toLow) численно могут быть больше верхних пределов (fromHigh, toHigh). В этом случае функция map() может использоваться для создания обратного диапазона чисел, например:

```
y = map(x, 1, 50, 50, 1);
```

Функция может обрабатывать отрицательные числа, поэтому этот пример

```
y = map(x, 1, 50, 50, -100);
```

также работает корректно.

Функция map() использует целочисленные вычисления, поэтому не возвращает дробных значений, как это иногда ожидается. При этом дробная часть числа просто отбрасывается, без округления или вычисления средних значений.

Параметры

value: переменная, значение которой необходимо преобразовать

fromLow: нижний предел текущего диапазона переменной value

fromHigh: верхний предел текущего диапазона переменной value

toLow: нижний предел нового диапазона переменной value

toHigh: верхний предел нового диапазона переменной value

Возвращаемые значения

Преобразованное значение.

Пример

```
/* Преобразование аналогового значения в 8-битное число (от 0 до 255) */
void setup() {}

void loop()
{
  int val = analogRead(0);
  val = map(val, 0, 1023, 0, 255);
  analogWrite(9, val);
}
```

Дополнение

Для интересующихся математикой приводим исходный код функции:

```
long map(long x, long in_min, long in_max, long out_min, long out_max)
{
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

Смотрите также

constrain()

pow(base, exponent)

Описание

Вычисляет значение числа, возведенного в степень. Функция pow() может использоваться для возведения числа в дробную степень, что может быть полезно для экспоненциального представления чисел или кривых.

Параметры

base: число (*float*)

exponent: показатель степени, в которую необходимо возвести число (*float*)

Возвращаемые значения

Результат возведения в степень (*double*)

Пример

См. функцию fscale в библиотеке.

Смотрите также

sqrt()
float
double

sqrt(x)

Описание

Вычисляет квадратный корень числа.

Параметры

x: число, любой тип данных

Возвращаемые значения

double, квадратный корень числа.

Смотрите также

pow()
sq()

sq(x)

Описание

Вычисляет квадрат числа: число, умноженное само на себя.

Параметры

x: число, любой тип данных

Возвращаемые значения

квадрат числа

Смотрите также

pow()
sqrt()

sin(rad)

Описание

Вычисляет синус угла (в радианах). Результат лежит в пределах от -1 до 1.

Параметры

rad: угол в радианах (*float*)

Возвращаемые значения

синус угла (*double*)

Смотрите также

cos()
tan()
float
double

cos(rad)

Описание

Вычисляет косинус угла (в радианах). Результат лежит в пределах от -1 до 1.

Параметры

rad: угол в радианах (*float*)

Возвращаемые значения

косинус угла ("double")

Смотрите также

sin()
tan()
float
double

tan(rad)

Описание

Вычисляет тангенс угла (в радианах). Результат лежит в пределах от минус бесконечности до бесконечности.

Параметры

rad: угол в радианах (*float*)

Возвращаемые значения

тангенс угла (*double*)

Смотрите также

sin()
cos()
float
double

randomSeed(seed)

Описание

Функция randomSeed() инициализирует генератор псевдо-случайных чисел, запуская его с произвольной точки последовательности случайных чисел. Несмотря на то, что эта последовательность состоит из очень большого количества случайных чисел, она всегда одна и та же.

Если при циклическом использовании функции random() важно получать последовательность различных случайных чисел, то для инициализации генератора случайных чисел используйте функцию randomSeed() с параметром, представляющим собой в определенной степени случайную величину. Таким параметром может быть, например, величина напряжения, считанная функцией analogRead() с неподсоединенного вывода.

Изредка требуется и обратное - использовать точно повторяющиеся последовательности псевдо-случайных чисел. Для этого перед началом генерирования последовательности случайных чисел необходимо вызвать функцию randomSeed() с параметром, представляющим собой фиксированное число.

Параметры

long, int - число для генерирования начального значения.

Возвращаемые значения

нет

Пример

```
long randNumber;

void setup(){
  Serial.begin(9600);
  randomSeed(analogRead(0));
}

void loop(){
  randNumber = random(300);
  Serial.println(randNumber);

  delay(50);
```

```
}
```

Смотрите также

[random()]

random()

Описание

Функция random() генерирует псевдо-случайные числа.

Синтаксис

```
random(max)
random(min, max)
```

Параметры

min - нижняя граница случайной величины, включительно (необязательный параметр).

max - верхняя граница случайной величины, не включительно.

Возвращаемые значения

случайное число в диапазоне от min до max-1 (*long*)

Примечание

Если при циклическом использовании функции random() важно получать последовательность различных случайных чисел, то для инициализации генератора случайных чисел используйте функцию randomSeed() с параметром, представляющим собой в определенной степени случайную величину. Таким параметром может быть, например, величина напряжения, считанная функцией analogRead() с не подсоединенного вывода.

Изредка требуется и обратное - использовать точно повторяющиеся последовательности псевдо-случайных чисел. Для этого перед началом генерирования последовательности случайных чисел необходимо вызвать функцию randomSeed() с параметром, представляющим собой фиксированное число.

Пример

```
long randNumber;

void setup(){
  Serial.begin(9600);
```

```
  // если аналоговый вход 0 ни к чему не подсоединен, то
произвольный аналоговый
  // шум на нем обеспечит разные исходные числа, передаваемые
функции randomSeed()
  // при каждом запуске программы. Впоследствии это позволит
функции random
  // генерировать разные значения.
  randomSeed(analogRead(0));
}

void loop() {
  // выводим случайное число в диапазоне от 0 до 299
  randNumber = random(300);
  Serial.println(randNumber);

  // выводим случайное число в диапазоне от 10 до 19
  randNumber = random(10, 20);
  Serial.println(randNumber);

  delay(50);
}
```

Смотрите также

randomSeed()

owByte()

Описание

Извлекает младший (крайний правый) байт переменной (например, типа word).

Синтаксис

```
lowByte(x)
```

Параметры

x: значение любого типа

Возвращаемые значения

byte

Смотрите также

highByte()
word()

highByte()

Описание

Извлекает старший (крайний левый) байт переменной типа word (либо второй младший байт переменной, если ее тип занимает больше двух байт).

Синтаксис

```
highByte(x)
```

Параметры

x: значение любого типа

Возвращаемые значения

byte

Смотрите также

lowByte()
word()

bitRead()

Описание

Считывает состояние указанного бита числа.

Синтаксис

```
bitRead(x, n)
```

Параметры

x: число, у которого необходимо считать бит

n: номер бита, состояние которого необходимо считать. Нумерация начинается с младшего значащего бита (крайнего правого) с номером 0

Возвращаемые значения

Состояние бита (0 или 1)

Смотрите также

bit()
bitWrite()
bitSet()
bitClear()

bitWrite()

Описание

Изменяет состояние указанного бита переменной.

Синтаксис

```
bitWrite(x, n, b)
```

Параметры

x: числовая переменная, у которой необходимо изменить бит

n: номер бита, состояние которого необходимо изменить. Нумерация начинается с младшего значащего бита (крайнего правого) с номером 0

b: новое значение бита (0 или 1)

Возвращаемые значения

Нет

Смотри

bitSet()

Описание

Устанавливает указанный бит (записывает 1) числовой переменной.

Синтаксис

```
bitSet(x, n)
```

Параметры

x: числовая переменная, у которой необходимо установить бит

n: номер бита, который необходимо установить. Нумерация начинается с младшего значащего бита (крайнего правого) с номером 0

Возвращаемые значения

Нет

Смотрите также

bit()
bitRead()
bitWrite()
bitClear()

bitClear()

Описание

Сбрасывает указанный бит (записывает 0) числовой переменной.

Синтаксис

```
bitClear(x, n)
```

Параметры

x: числовая переменная, у которой необходимо сбросить бит

n: номер бита, который необходимо сбросить. Нумерация начинается с младшего значащего бита (крайнего правого) с номером 0

Возвращаемые значения

Нет

Смотрите также

bit()
bitRead()
bitWrite()
bitSet()

bit()

Описание

Вычисляет значение указанного бита (бит 0 - 1, бит 1 - 2, бит 2 - 4, и т.д.).

Синтаксис

```
bit(n)
```

Параметры

n: бит, значение которого необходимо вычислить.

Возвращаемые значения

Значение бита

Смотрите также

bitRead()
bitWrite()
bitSet()
bitClear()

attachInterrupt()

Описание

Задает функцию, которую необходимо вызвать при возникновении внешнего прерывания. Заменяет предыдущую функцию, если таковая была ранее ассоциирована с прерыванием. В большинстве плат Ардуино существует два внешних прерывания: номер 0 (цифровой вывод 2) и 1 (цифровой вывод 3). Номера выводов для внешних прерываний, доступные в тех или иных платах Ардуино, приведены в таблице ниже:

Плата	**int.0**	**int.1**	**int.2**	**int.3**	**int.4**	**int.5**
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
Leonardo	3	2	0	1	7	
Due	**(см. ниже)**					

Плата Arduino Due предоставляет больше возможностей по работе с прерываниями, поскольку позволяют ассоциировать функцию-обработчик прерывания с любым из доступных выводов. При этом в функции attachInterrupt() можно непосредственно указывать номер вывода.

Синтаксис

```
attachInterrupt(interrupt, function, mode)
attachInterrupt(pin, function, mode)     (только для Arduino Due)
```

Параметры

interrupt:	номер прерывания (*int*)
pin:	номер вывода (*только для Arduino Due*)
function:	функция, которую необходимо вызвать при возникновении прерывания; эта функция должна быть без параметров и не возвращать никаких значений. Такую функцию иногда называют *обработчиком прерывания*.

mode:	определяет условие, при котором должно срабатывать прерывание. Может принимать одно из четырех предопределенных значений: **LOW** - прерывание будет срабатывать всякий раз, когда на выводе присутствует низкий уровень сигнала **CHANGE** - прерывание будет срабатывать всякий раз, когда меняется состояние вывода **RISING** - прерывание сработает, когда состояние вывода изменится с низкого уровня на высокий **FALLING** - прерывание сработает, когда состояние вывода изменится с высокого уровня на низкий. В Arduino Due доступно еще одно значение: **HIGH** - прерывание будет срабатывать всякий раз, когда на выводе присутствует высокий уровень сигнала (*только для Arduino Due*).

Возвращаемые значения

нет

Примечание

Внутри функции-обработчика прерывания функция delay() не будет работать; значения, возвращаемые функцией millis(), не будут увеличиваться. Также будут потеряны данные, полученные по последовательному интерфейсу во время выполнения обработчика прерывания. Любые переменные, которые изменяются внутри функции обработчика должны быть объявлены как volatile.

Использование прерываний

Прерывания могут использоваться для того, чтобы заставить микроконтроллер выполнять определенные участки программы автоматически и позволяют решить проблемы с синхронизацией. Типичными задачами, требующими использования прерываний, являются считывание сигнала энкодера (датчика вращения) либо мониторинг воздействий пользователя.

В программе, от которой требуется постоянно обрабатывать сигнал от датчика вращения (не пропуская при этом ни единого импульса), очень сложно реализовать еще какую-либо функциональность помимо опроса. Это объясняется тем, что для своевременной обработки поступающих импульсов, программа должна постоянно опрашивать выводы, к которым подключен энкодер. При работе с другими типами датчиков часто требуется подобный динамический интерфейс: например, при опросе звукового датчика с целью различить щелчок, или при работе с инфракрасным щелевым датчиком (фото-прерывателем) для распознавания момента броска монеты. Во всех этих примерах использование прерываний позволит разгрузить микроконтроллер для выполнения другой работы, при этом не теряя контроль над поступающими сигналами.

Пример

```
int pin = 13;
volatile int state = LOW;

void setup()
{
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);
}

void loop()
{
  digitalWrite(pin, state);
}

void blink()
{
  state = !state;
}
```

Смотрите также

detachInterrupt()

detachInterrupt()

Описание

Запрещает заданное прерывание.

Синтаксис

```
detachInterrupt(interrupt)
detachInterrupt(pin)   (только для Arduino Due)
```

Параметры

interrupt: номер прерывания, которое необходимо запретить (подробнее см. attachInterrupt()).
pin: номер вывода, соответствующее прерывание которого необходимо запретить (*только для Arduino Due*)

Смотрите также

attachInterrupt()

interrupts()

Описание

Повторно разрешает прерывания (после того, как они были отключены функцией noInterrupts()). Прерывания позволяют некоторым важным задачам выполняться в фоновом режиме и по умолчанию включены. Если прерывания отключены, некоторые функции не будут работать, а поступающие от других устройств данные могут игнорироваться. Однако, прерывания могут незначительно замедлять выполнение программы, поэтому в наиболее критичных ко времени участках кода они могут быть отключены.

Параметры

Нет

Возвращаемые значения

Нет

Пример

```
void setup() {}

void loop()
{
  noInterrupts();
  // критичный ко времени участок кода
  interrupts();
  // далее - остальная программа
}
```

Смотрите также

noInterrupts()
attachInterrupt()
detachInterrupt()

noInterrupts()

Описание

Запрещает прерывания (повторно разрешить их можно функцией interrupts()). Прерывания позволяют некоторым важным задачам выполняться в фоновом режиме и по умолчанию включены. Если прерывания отключены, некоторые функции не будут работать, а поступающие от других устройств данные могут игнорироваться. Однако, прерывания могут незначительно замедлять выполнение программы, поэтому в наиболее критичных ко времени участках кода они могут быть отключены.

Параметры

Нет

Возвращаемые значения

Нет

Пример

```
void setup() {}

void loop()
{
  noInterrupts();
  // критичный ко времени участок кода
  interrupts();
  // далее - остальная программа
}
```

Смотрите также

interrupts()