ARDUINO

Language Reference

Hazırlayan:

Fazıl Demir

elobilgi.com

V1 - Nisan-2018

Kaynak:

https://www.arduino.cc/reference/en/

Arduino

Language Reference

Arduino programming language can be divided in three main parts: structure, values (variables and constants), and functions.

Structure

The elements of Arduino (C++) code.

Functions

For controlling the Arduino board and performing computations.

Structure

setup() [Sketch]

Description

The `setup()` function is called when a sketch starts. Use it to initialize variables, pin modes, start using libraries, etc. The `setup()` function will only run once, after each powerup or reset of the Arduino board.

Example Code

```
int buttonPin = 3;

void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}

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

loop() [Sketch]

Description

After creating a setup() function, which initializes and sets the initial values, the `loop()` function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond. Use it to actively control the Arduino board.

Example Code

```
int buttonPin = 3;

// setup initializes serial and the button pin
void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}

// loop checks the button pin each time,
// and will send serial if it is pressed
void loop()
{
  if (digitalRead(buttonPin) == HIGH)
    Serial.write('H');
  else
    Serial.write('L');

  delay(1000);
}
```

if...else [Control Structure]

Description

The `if` statement checks for a condition and executes the proceeding statement or set of statements if the condition is 'true'.

Syntax

```
if (condition)
{
  //statement(s)
}
```

Parameters

condition: a boolean expression i.e., can be `true` or `false`

Example Code

The brackets may be omitted after an if statement. If this is done, the next line (defined by the semicolon) becomes the only conditional statement.

```
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);
}                                  // all are correct
```

Notes and Warnings

The statements being evaluated inside the parentheses require the use of one or more operators shown below.

Comparison Operators:

```
x == y (x is equal to y)
x != y (x is not equal to y)
x <  y (x is less than y)
x >  y (x is greater than y)
x <= y (x is less than or equal to y)
x >= y (x is greater than or equal to y)
```

Beware of accidentally using the single equal sign (e.g. `if (x = 10)`). The single equal sign is the assignment operator, and sets `x` to 10 (puts the value 10 into the variable `x`). Instead use the double equal sign (e.g. `if (x == 10)`), which is the comparison operator, and tests *whether* `x` is equal to 10 or not. The latter statement is only true if `x` equals 10, but the former statement will always be true.

This is because C evaluates the statement `if (x=10)` as follows: 10 is assigned to `x` (remember that the single equal sign is the (assignment operator)), so `x` now contains 10. Then the 'if' conditional evaluates 10, which always evaluates to `TRUE`, since any non-zero number evaluates to TRUE. Consequently, `if (x = 10)` will always evaluate to `TRUE`, which is not the desired result when using an 'if' statement. Additionally, the variable `x` will be set to 10, which is also not a desired action.

else [Control Structure]

Description

The `if…else` allows greater control over the flow of code than the basic if statement, by allowing multiple tests to be grouped together. An `else` clause (if at all exists) will be executed if the condition in the `if` statement results in `false`. The `else` can proceed another `if` test, so that multiple, mutually exclusive tests can be run at the same time.

Each test will proceed to the next one until a true test is encountered. When a true test is found, its associated block of code is run, and the program then skips to the line following the entire if/else construction. If no test proves to be true, the default `else` block is executed, if one is present, and sets the default behavior.

Note that an `else if` block may be used with or without a terminating `else` block and vice versa. An unlimited number of such `else if` branches is allowed.

Syntax

```
if (condition1)
{
  // do Thing A
}
else if (condition2)
{
  // do Thing B
}
else
{
  // do Thing C
}
```

Example Code

Below is an extract from a code for temperature sensor system

```
if (temperature >= 70)
{
  //Danger! Shut down the system
}
else if (temperature >= 60 && temperature < 70)
{
  //Warning! User attention required
}
else
{
  //Safe! Continue usual tasks...
}
```

for [Control Structure]

Description

The `for` statement is used to repeat a block of statements enclosed in curly braces. An increment counter is usually used to increment and terminate the loop. The `for` statement is useful for any repetitive operation, and is often used in combination with arrays to operate on collections of data/pins.

Syntax

```
for (initialization; condition; increment) {
        //statement(s);
}
```

The **initialization** happens first and exactly once. Each time through the loop, the **condition** is tested; if it's `true`, the statement block, and the **increment** is executed, then the **condition** is tested again. When the **condition** becomes `false`, the loop ends.

Example Code

```
// Dim an LED using a PWM pin
int PWMpin = 10; // LED in series with 470 ohm resistor on pin 10

void setup()
{
  // no setup needed
}

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

Notes and Warnings

The C `for` loop is much more flexible than `for` loops found in some other computer languages, including BASIC. Any or all of the three header elements may be omitted, although the semicolons are required. Also the statements for initialization, condition, and increment can be any valid C statements with unrelated variables, and use any C datatypes including floats. These types of unusual `for` statements may provide solutions to some rare programming problems.

For example, using a multiplication in the increment line will generate a logarithmic progression:

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

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

Another example, fade an LED up and down with one `for` loop:

```
void loop()
{
   int x = 1;
   for (int i = 0; i > -1; i = i + x){
      analogWrite(PWMpin, i);
      if (i == 255) x = -1;             // switch direction at peak
      delay(10);
   }
}
```

switch...case [Control Structure]

Description

Like if statements, switch case controls the flow of programs by allowing programmers to specify different code that should be executed in various conditions. In particular, a switch statement compares the value of a variable to the values specified in case statements. When a case statement is found whose value matches that of the variable, the code in that case statement is run.

The break keyword exits the switch statement, and is typically used at the end of each case. Without a break statement, the switch statement will continue executing the following expressions ("falling-through") until a break, or the end of the switch statement is reached.

Syntax

```
switch (var) {
  case label1:
    // statements
    break;
  case label2:
    // statements
    break;
  default:
    // statements
}
```

Parameters

`var`: a variable whose value to compare with various cases. **Allowed data types:** int, char
`label1`, `label2`: constants. **Allowed data types:** int, char

Returns

Nothing

Example Code

```
  switch (var) {
    case 1:
      //do something when var equals 1
      break;
    case 2:
      //do something when var equals 2
      break;
    default:
      // if nothing else matches, do the default
      // default is optional
      break;
  }
```

while [Control Structure]

Description

A `while` loop will loop continuously, and infinitely, until the expression inside the parenthesis, () becomes false. Something must change the tested variable, or the while loop will never exit. This could be in your code, such as an incremented variable, or an external condition, such as testing a sensor.

Syntax

```
while(condition){
  // statement(s)
}
```

The `condition` is a boolean expression that evaluates to `true` or `false`.

Example Code

```
var = 0;
while(var < 200){
  // do something repetitive 200 times
  var++;
}
```

do...while [Control Structure]

Description

The `do…while` loop works in the same manner as the while loop, with the exception that the condition is tested at the end of the loop, so the do loop will always run at least once.

Syntax

```
do
{
    // statement block
} while (condition);
```

The `condition` is a boolean expression that evaluates to `true` or `false`.

Example Code

```
do
{
  delay(50);          // wait for sensors to stabilize
  x = readSensors();  // check the sensors

} while (x < 100);
```

break [Control Structure]

Description

`break` is used to exit from a for, while or do…while loop, bypassing the normal loop condition. It is also used to exit from a switch case statement.

Example Code

In the following code, the control exits the `for` loop when the sensor value exceeds the threshold.

```
for (x = 0; x < 255; x ++)
{
    analogWrite(PWMpin, x);
    sens = analogRead(sensorPin);
    if (sens > threshold){      // bail out on sensor detect
       x = 0;
       break;
    }
    delay(50);
}
```

continue [Control Structure]

Description

The `continue` statement skips the rest of the current iteration of a loop (for, while, or do…while). It continues by checking the conditional expression of the loop, and proceeding with any subsequent iterations.

Example Code

The following code writes the value of 0 to 255 to the `PWMpin`, but skips the values in the range of 41 to 119.

```
for (x = 0; x <= 255; x ++)
{
    if (x > 40 && x < 120){      // create jump in values
        continue;
    }

    analogWrite(PWMpin, x);
    delay(50);
}
```

return [Control Structure]

Description

Terminate a function and return a value from a function to the calling function, if desired.

Syntax

```
return;

return value; // both forms are valid
```

Parameters

`value': any variable or constant type

Example Code

A function to compare a sensor input to a threshold

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

The return keyword is handy to test a section of code without having to "comment out" large sections of possibly buggy code.

```
void loop(){
// brilliant code idea to test here

return;

// the rest of a dysfunctional sketch here
// this code will never be executed
}
```

goto [Control Structure]

Description

Transfers program flow to a labeled point in the program

Syntax

```
label:

goto label; // sends program flow to the label
```

Example Code

```
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;}
            // more statements ...
        }
    }
}

bailout:
```

Notes and Warnings

The use of `goto` is discouraged in C programming, and some authors of C programming books claim that the `goto` statement is never necessary, but used judiciously, it can simplify certain programs. The reason that many programmers frown upon the use of goto is that with the unrestrained use of `goto` statements, it is easy to create a program with undefined program flow, which can never be debugged.

With that said, there are instances where a `goto` statement can come in handy, and simplify coding. One of these situations is to break out of deeply nested for loops, or if logic blocks, on a certain condition.

; [Further Syntax]

Description

Used to end a statement.

Example Code

```
int a = 13;
```

Notes and Warnings

Forgetting to end a line in a semicolon will result in a compiler error. The error text may be obvious, and refer to a missing semicolon, or it may not. If an impenetrable or seemingly illogical compiler error comes up, one of the first things to check is a missing semicolon, in the immediate vicinity, preceding the line at which the compiler complained.

{} [Further Syntax]

Description

Curly braces (also referred to as just "braces" or as "curly brackets") are a major part of the C programming language. They are used in several different constructs, outlined below, and this can sometimes be confusing for beginners.
An opening curly brace `{` must always be followed by a closing curly brace `}`. This is a condition that is often referred to as the braces being balanced. The Arduino IDE (Integrated Development Environment) includes a convenient feature to check the balance of curly braces. Just select a brace, or even click the insertion point immediately following a brace, and its logical companion will be highlighted.

Beginners programmers, and programmers coming to C from the BASIC language often find using braces confusing or daunting. After all, the same curly braces replace the RETURN statement in a subroutine (function), the ENDIF statement in a conditional and the NEXT statement in a FOR loop.

Unbalanced braces can often lead to cryptic, impenetrable compiler errors that can sometimes be hard to track down in a large program. Because of their varied usages, braces are also incredibly important to the syntax of a program and moving a brace one or two lines will often dramatically affect the meaning of a program.

Example Code

The main uses of curly braces are listed in the examples below.

Functions

```
void myfunction(datatype argument){
  statements(s)
}
```

Loops

```
while (boolean expression)
{
 statement(s)
}
do
{
 statement(s)
} while (boolean expression);

for (initialisation; termination condition; incrementing expr)
{
 statement(s)
}
```

Conditional Statements

```
if (boolean expression){
 statement(s)
}
else if (boolean expression){
 statement(s)
}else{
 statement(s)
}
```

// [Further Syntax]

Description

Comments are lines in the program that are used to inform yourself or others about the way the program works. They are ignored by the compiler, and not exported to the processor, so they don't take up any space in the microcontroller's flash memory. Comments' only purpose is to help you understand (or remember), or to inform others about how your program works.

A **single line comment** begins with `//` (two adjacent slashes). This comment ends automatically at the end of a line. whatever follows `//` till the end of a line will be ignored by the compiler.

Example Code

There are two different ways of marking a line as a comment:

```
// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;


digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
```

Notes and Warnings

When experimenting with code, "commenting out" parts of your program is a convenient way to remove lines that may be buggy. This leaves the lines in the code, but turns them into comments, so the compiler just ignores them. This can be especially useful when trying to locate a problem, or when a program refuses to compile and the compiler error is cryptic or unhelpful.

/* */ [Further Syntax]

Description

Comments are lines in the program that are used to inform yourself or others about the way the program works. They are ignored by the compiler, and not exported to the processor, so they don't take up any space in the microcontroller's flash memory. Comments' only purpose is to help you understand (or remember), or to inform others about how your program works.

The beginning of a **block comment** or a **multi-line comment** is marked by the symbol `/*` and the symbol `*/` marks its end. This type of a comment is called so as this can extend over more than one line; once the compiler reads the `/*` it ignores whatever follows unitl it enounters a `*/`.

Example Code

```
/* This is a valid comment */

/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
  (Another valid comment)
*/

/*
  if (gwb == 0){   // single line comment is OK inside a multi-line comment
  x = 3;           /* but not another multi-line comment - this is invalid */
  }
// don't forget the "closing" comment - they have to be balanced!
*/
```

Notes and Warnings

When experimenting with code, "commenting out" parts of your program is a convenient way to remove lines that may be buggy. This leaves the lines in the code, but turns them into comments, so the compiler just ignores them. This can be especially useful when trying to locate a problem, or when a program refuses to compile and the compiler error is cryptic or unhelpful.

#define [Further Syntax]

Description

`#define` is a useful C component that allows the programmer to give a name to a constant value before the program is compiled. Defined constants in arduino don't take up any program memory space on the chip. The compiler will replace references to these constants with the defined value at compile time.

This can have some unwanted side effects though, if for example, a constant name that had been #defined is included in some other constant or variable name. In that case the text would be replaced by the #defined number (or text).

In general, the const keyword is preferred for defining constants and should be used instead of #define.

Syntax

```
#define constantName value
```

Note that the # is necessary.

Example Code

```
#define ledPin 3
// The compiler will replace any mention of ledPin with the value 3 at compile
time.
```

Notes and Warnings

There is no semicolon after the #define statement. If you include one, the compiler will throw cryptic errors further down the page.

```
#define ledPin 3;    // this is an error
```

Similarly, including an equal sign after the #define statement will also generate a cryptic compiler error further down the page.

```
#define ledPin  = 3  // this is also an error
```

#include [Further Syntax]

Description

`#include` is used to include outside libraries in your sketch. This gives the programmer access to a large group of standard C libraries (groups of pre-made functions), and also libraries written especially for Arduino.

The main reference page for AVR C libraries (AVR is a reference to the Atmel chips on which the Arduino is based) is here.

Note that `#include`, similar to `#define`, has no semicolon terminator, and the compiler will yield cryptic error messages if you add one.

Example Code

This example includes a library that is used to put data into the program space *flash* instead of *ram*. This saves the ram space for dynamic memory needs and makes large lookup tables more practical.

```
#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};
```

= [Arithmetic Operators]

Description

The single equal sign = in the C programming language is called the assignment operator. It has a different meaning than in algebra class where it indicated an equation or equality. The assignment operator tells the microcontroller to evaluate whatever value or expression is on the right side of the equal sign, and store it in the variable to the left of the equal sign.

Example Code

```
int sensVal;               // declare an integer variable named sensVal
sensVal = analogRead(0);  // store the (digitized) input voltage at analog pin 0
in SensVal
```

Notes and Warnings

1. The variable on the left side of the assignment operator (= sign) needs to be able to hold the value stored in it. If it is not large enough to hold a value, the value stored in the variable will be incorrect.
2. Don't confuse the assignment operator [=] (single equal sign) with the comparison operator [==] (double equal signs), which evaluates whether two expressions are equal.

+ [Arithmetic Operators]

Description

Addition is one of the four primary arithmetic operations. The operator + (plus) operates on two operands to produce the sum.

Syntax

```
sum = operand1 + operand2;
```

Parameters

`sum` : variable. **Allowed data types:** int, float, double, byte, short, long
`operand1` : variable or constant. **Allowed data types:** int, float, double, byte, short, long
`operand2` : variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
int a = 5, b = 10, c = 0;
c = a + b; // the variable 'c' gets a value of 15 after this statement is
executed
```

Notes and Warnings

1. The addition operation can overflow if the result is larger than that which can be stored in the data type (e.g. adding 1 to an integer with the value 32,767 gives -32,768).
2. If one of the numbers (operands) are of the type float or of type double, floating point math will be used for the calculation.
3. If the operands are of float / double data type and the variable that stores the sum is an integer, then only the integral part is stored and the fractional part of the number is lost.

```
float a = 5.5, b = 6.6;
int c = 0;
c = a + b; // the variable 'c' stores a value of 12 only as opposed to the
expected sum of 12.1
```

- [Arithmetic Operators]

Description

Subtraction is one of the four primary arithmetic operations. The operator - (minus) operates on two operands to produce the difference of the second from the first.

Syntax

```
difference = operand1 - operand2;
```

Parameters

`difference` : variable. **Allowed data types:** int, float, double, byte, short, long
`operand1` : variable or constant. **Allowed data types:** int, float, double, byte, short, long
`operand2` : variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
int a = 5, b = 10, c = 0;
c = a - b; // the variable 'c' gets a value of -5 after this statement is
executed
```

Notes and Warnings

1. The subtraction operation can overflow if the result is smaller than that which can be stored in the data type (e.g. subtracting 1 from an integer with the value -32,768 gives 32,767).
2. If one of the numbers (operands) are of the type float or of type double, floating point math will be used for the calculation.
3. If the operands are of float / double data type and the variable that stores the difference is an integer, then only the integral part is stored and the fractional part of the number is lost.

```
float a = 5.5, b = 6.6;
int c = 0;
c = a - b; // the variable 'c' stores a value of -1 only as opposed to the
expected difference of -1.1
```

* [Arithmetic Operators]

Description

Multiplication is one of the four primary arithmetic operations. The operator `*` (asterisk) operates on two operands to produce the product.

Syntax

```
product = operand1 * operand2;
```

Parameters

`product` : variable. **Allowed data types:** int, float, double, byte, short, long
`operand1` : variable or constant. **Allowed data types:** int, float, double, byte, short, long
`operand2` : variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
int a = 5, b = 10, c = 0;
c = a * b; // the variable 'c' gets a value of 50 after this statement is
executed
```

Notes and Warnings

1. The multiplication operation can overflow if the result is bigger than that which can be stored in the data type.
2. If one of the numbers (operands) are of the type float or of type double, floating point math will be used for the calculation.
3. If the operands are of float / double data type and the variable that stores the product is an integer, then only the integral part is stored and the fractional part of the number is lost.

```
float a = 5.5, b = 6.6;
int c = 0;
c = a * b; // the variable 'c' stores a value of 36 only as opposed to the
expected product of 36.3
```

/ [Arithmetic Operators]

Description

Division is one of the four primary arithmetic operations. The operator `/` (slash) operates on two operands to produce the result.

Syntax

```
result = numerator / denominator;
```

Parameters

`result` : variable. **Allowed data types:** int, float, double, byte, short, long
`numerator` : variable or constant. **Allowed data types:** int, float, double, byte, short, long
`denominator` : **non zero** variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
int a = 50, b = 10, c = 0;
c = a / b; // the variable 'c' gets a value of 5 after this statement is
executed
```

Notes and Warnings

1. If one of the numbers (operands) are of the type float or of type double, floating point math will be used for the calculation.
2. If the operands are of float / double data type and the variable that stores the sum is an integer, then only the integral part is stored and the fractional part of the number is lost.

```
float a = 55.5, b = 6.6;
int c = 0;
c = a / b; // the variable 'c' stores a value of 8 only as opposed to the
expected result of 8.409
```

%   [Arithmetic Operators]

Description

Modulo operation calculates the remainder when one integer is divided by another. It is useful for keeping a variable within a particular range (e.g. the size of an array). The `%` (percent) symbol is used to carry out modulo operation.

Syntax

```
remainder = dividend % divisor;
```

Parameters

`remainder` : variable. **Allowed data types:** int, float, double
`dividend` : variable or constant. **Allowed data types:** int
`divisor` : **non zero** variable or constant. **Allowed data types:** int

Example Code

```
int x = 0;
x = 7 % 5;    // x now contains 2
x = 9 % 5;    // x now contains 4
x = 5 % 5;    // x now contains 0
x = 4 % 5;    // x now contains 4
/* update one value in an array each time through a loop */

int values[10];
int i = 0;

void setup() {}

void loop()
{
  values[i] = analogRead(0);
  i = (i + 1) % 10;   // modulo operator rolls over variable
}
```

Notes and Warnings

The modulo operator does not work on floats.

==   [Comparison Operators]

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the two operands are equal. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

```
x == y;   // is true if x is equal to y and it is false if x is not equal to y
```

Parameters

x: variable. **Allowed data types:** int, float, double, byte, short, long
y: variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
if (x==y)      // tests if x is equal to y
{
// do something only if the comparison result is true
}
```

!=   [Comparison Operators]

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the two operands are not equal. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

```
x != y;   // is false if x is equal to y and it is true if x is not equal to y
```

Parameters

x: variable. **Allowed data types:** int, float, double, byte, short, long
y: variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
if (x!=y)      // tests if x is not equal to y
{
// do something only if the comparison result is true
}
```

< [Comparison Operators]

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is less (smaller) than the operand on the right. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

```
x < y;   // is true if x is smaller than y and it is false if x is equal or
bigger than y
```

Parameters

`x`: variable. **Allowed data types:** int, float, double, byte, short, long
`y`: variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
if (x<y)      // tests if x is less (smaller) than y
{
// do something only if the comparison result is true
}
```

Notes and Warnings

Negative numbers are less than positive numbers.

> [Comparison Operators]

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is greater (bigger) than the operand on the right. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

```
x > y;   // is true if x is bigger than y and it is false if x is equal or
smaller than y
```

Parameters

`x`: variable. **Allowed data types:** int, float, double, byte, short, long
`y`: variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
if (x>y)      // tests if x is greater (bigger) than y
{
// do something only if the comparison result is true
}
```

Notes and Warnings

Positive numbers are greater than negative numbers.

<=   [Comparison Operators]

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is less (smaller) than or equal to the operand on the right. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

```
x <= y;   // is true if x is smaller than or equal to y and it is false if x is
greater than y
```

Parameters

`x`: variable. **Allowed data types:** int, float, double, byte, short, long
`y`: variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
if (x<=y)      // tests if x is less (smaller) than or equal to y
{
// do something only if the comparison result is true
}
```

Notes and Warnings

Negative numbers are smaller than positive numbers.

>=   [Comparison Operators]

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is greater (bigger) than or equal to the operand on the right. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

```
x >= y;   // is true if x is bigger than or equal to y and it is false if x is
smaller than y
```

Parameters

`x`: variable. **Allowed data types:** int, float, double, byte, short, long
`y`: variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
if (x>=y)      // tests if x is greater (bigger) than or equal to y
{
// do something only if the comparison result is true
}
```

Notes and Warnings

Positive numbers are greater than negative numbers.

&& [Boolean Operators]

Description

Logical AND results in `true` **only** if both operands are `true`.

Example Code

This operator can be used inside the condition of an if statement.

```
if (digitalRead(2) == HIGH  && digitalRead(3) == HIGH) { // if BOTH the switches
read HIGH
    // statements
}
```

Notes and Warnings

Make sure you don’t mistake the boolean AND operator, && (double ampersand) for the bitwise AND operator & (single ampersand). They are entirely different beasts.

|| [Boolean Operators]

Description

Logical OR results in a `true` if either of the two operands is `true`.

Example Code

This operator can be used inside the condition of an if statement.

```
if (x > 0 || y > 0) { // if either x or y is greater than zero
  // statements
}
```

Notes and Warnings

Do not confuse the boolean || (double pipe) operator with the bitwise OR operator | (single pipe).

! [Boolean Operators]

Description

Logical NOT results in a `true` if the operand is `false` and vice versa.

Example Code

This operator can be used inside the condition of an if statement.

```
if (!x) { // if x is not true
  // statements
}
```

It can be used to invert the boolean value.

```
x = !y;   // the inverted value of y is stored in x
```

Notes and Warnings

The bitwise not ~ (tilde) looks much different than the boolean not ! (exclamation point or "bang" as the programmers say) but you still have to be sure which one you want where.

* [Pointer Access Operators]

Description

Dereferencing is one of the features specifically for use with pointers. The asterisk operator `*` is used for this purpose. If `p` is a pointer, then `*p` represents the value contained in the address pointed by `p`.

Example Code

```
int *p;        // declare a pointer to an int data type
int i = 5, result = 0;
p = &i;        // now 'p' contains the address of 'i'
result = *p;   // 'result' gets the value at the address pointed by 'p'
               // i.e., it gets the value of 'i' which is 5
```

Notes and Warnings

Pointers are one of the complicated subjects for beginners in learning C, and it is possible to write the vast majority of Arduino sketches without ever encountering pointers. However for manipulating certain data structures, the use of pointers can simplify the code, and and knowledge of manipulating pointers is handy to have in one’s toolkit.

& [Pointer Access Operators]

Description

Referencing is one of the features specifically for use with pointers. The ampersand operator `&` is used for this purpose. If `x` is a variable, then `&x` represents the address of the variable `x`.

Example Code

```
int *p;        // declare a pointer to an int data type
int i = 5, result = 0;
p = &i;        // now 'p' contains the address of 'i'
result = *p;   // 'result' gets the value at the address pointed by 'p'
               // i.e., it gets the value of 'i' which is 5
```

Notes and Warnings

Pointers are one of the complicated subjects for beginners in learning C, and it is possible to write the vast majority of Arduino sketches without ever encountering pointers. However for manipulating certain data structures, the use of pointers can simplify the code, and knowledge of manipulating pointers is handy to have in one’s toolkit.

& [Bitwise Operators]

Description

The bitwise AND operator in C++ is a single ampersand `&`, used between two other integer expressions. Bitwise AND operates on each bit position of the surrounding expressions independently, according to this rule: if both input bits are 1, the resulting output is 1, otherwise the output is 0.

Another way of expressing this is:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  0  0  1    (operand1 & operand2) - returned result
```

In Arduino, the type int is a 16-bit value, so using & between two int expressions causes 16 simultaneous AND operations to occur.

Example Code

In a code fragment like:

```
int a =  92;    // in binary: 0000000001011100
int b = 101;    // in binary: 0000000001100101
int c = a & b;  // result:    0000000001000100, or 68 in decimal.
```

Each of the 16 bits in a and b are processed by using the bitwise AND, and all 16 resulting bits are stored in c, resulting in the value 01000100 in binary, which is 68 in decimal.

One of the most common uses of bitwise AND is to select a particular bit (or bits) from an integer value, often called masking. See below for an example

```
PORTD = PORTD & B00000011;  // clear out bits 2 - 7, leave pins 0 and 1
untouched (xx & 11 == xx)
```

| [Bitwise Operators]

Description

The bitwise OR operator in C++ is the vertical bar symbol, |. Like the & operator, | operates independently each bit in its two surrounding integer expressions, but what it does is different (of course). The bitwise OR of two bits is 1 if either or both of the input bits is 1, otherwise it is 0.

In other words:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  1    (operand1 | operand2) - returned result
```

Example Code

```
int a =  92;    // in binary: 0000000001011100
int b = 101;    // in binary: 0000000001100101
int c = a | b;  // result:    0000000001111101, or 125 in decimal.
```

One of the most common uses of the Bitwise OR is to set multiple bits in a bit-packed number.

```
DDRD = DDRD | B11111100; // set direction bits for pins 2 to 7, leave 0 and 1
untouched (xx | 00 == xx)
// same as pinMode(pin, OUTPUT) for pins 2 to 7
```

^ [Bitwise Operators]

Description

There is a somewhat unusual operator in C++ called bitwise EXCLUSIVE OR, also known as bitwise XOR. (In English this is usually pronounced "eks-or".) The bitwise XOR operator is written using the caret symbol ^. A bitwise XOR operation results in a 1 only if the input bits are different, else it results in a 0.

Precisely,

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  0    (operand1 ^ operand2) - returned result
```

Example Code

```
int x = 12;     // binary: 1100
int y = 10;     // binary: 1010
int z = x ^ y;  // binary: 0110, or decimal 6
```

The ^ operator is often used to toggle (i.e. change from 0 to 1, or 1 to 0) some of the bits in an integer expression. In a bitwise XOR operation if there is a 1 in the mask bit, that bit is inverted; if there is a 0, the bit is not inverted and stays the same. Below is a program to blink digital pin 5.

```
// Blink_Pin_5
// demo for Exclusive OR
void setup(){
DDRD = DDRD | B00100000; // set digital pin five as OUTPUT
Serial.begin(9600);
}

void loop(){
PORTD = PORTD ^ B00100000;  // invert bit 5 (digital pin 5), leave others
untouched
delay(100);
}
```

~ [Bitwise Operators]

Description

The bitwise NOT operator in C++ is the tilde character ~. Unlike & and |, the bitwise NOT operator is applied to a single operand to its right. Bitwise NOT changes each bit to its opposite: 0 becomes 1, and 1 becomes 0.

In other words:

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

Example Code

```
int a = 103;     // binary:  0000000001100111
int b = ~a;      // binary:  1111111110011000 = -104
```

Notes and Warnings

You might be surprised to see a negative number like -104 as the result of this operation. This is because the highest bit in an int variable is the so-called sign bit. If the highest bit is 1, the number is interpreted as negative. This encoding of positive and negative numbers is referred to as two's complement. For more information, see the Wikipedia article on two's complement.

As an aside, it is interesting to note that for any integer x, ~x is the same as -x-1.

At times, the sign bit in a signed integer expression can cause some unwanted surprises.

<<   [Bitwise Operators]

Description

The left shift operator << causes the bits of the left operand to be shifted **left** by the number of positions specified by the right operand.

Syntax

```
variable << number_of_bits;
```

Parameters

`variable`: **Allowed data types:** byte, int, long
`number_of_bits`: a number that is < = 32. **Allowed data types:** int

Example Code

```
int a = 5;        // binary: 0000000000000101
int b = a << 3;   // binary: 0000000000101000, or 40 in decimal
```

Notes and Warnings

When you shift a value x by y bits (x << y), the leftmost y bits in x are lost, literally shifted out of existence:

```
int x = 5;        // binary: 0000000000000101
int y = 14;
int result = x << y;  // binary: 0100000000000000 - the first 1 in 101 was
discarded
```

If you are certain that none of the ones in a value are being shifted into oblivion, a simple way to think of the left-shift operator is that it multiplies the left operand by 2 raised to the right operand power. For example, to generate powers of 2, the following expressions can be employed:

```
   Operation  Result
   ---------  ------
    1 <<  0      1
    1 <<  1      2
    1 <<  2      4
    1 <<  3      8
    ...
    1 <<  8    256
    1 <<  9    512
    1 << 10   1024
    ...
```

The following example can be used to print out the value of a received byte to the serial monitor, using the left shift operator to move along the byte from bottom(LSB) to top (MSB), and print out its Binary value:

```
// Prints out Binary value (1 or 0) of byte
void printOut1(int c) {
  for (int bits = 7; bits > -1; bits--) {
    // Compare bits 7-0 in byte
    if (c & (1 << bits)) {
      Serial.print ("1");
    }
    else {
      Serial.print ("0");
    }
  }
}
```

>> [Bitwise Operators]

Description

The right shift operator >> causes the bits of the left operand to be shifted **right** by the number of positions specified by the right operand.

Syntax

```
variable >> number_of_bits;
```

Parameters

`variable`: **Allowed data types:** byte, int, long
`number_of_bits`: a number that is < = 32. **Allowed data types:** int

Example Code

```
int a = 40;        // binary: 0000000000101000
int b = a >> 3;    // binary: 0000000000000101, or 5 in decimal
```

Notes and Warnings

When you shift x right by y bits (x >> y), and the highest bit in x is a 1, the behavior depends on the exact data type of x. If x is of type int, the highest bit is the sign bit, determining whether x is negative or not, as we have discussed above. In that case, the sign bit is copied into lower bits, for esoteric historical reasons:

```
int x = -16;      // binary: 1111111111110000
int y = 3;
int result = x >> y;  // binary: 1111111111111110
```

This behavior, called sign extension, is often not the behavior you want. Instead, you may wish zeros to be shifted in from the left. It turns out that the right shift rules are different for unsigned int expressions, so you can use a typecast to suppress ones being copied from the left:

```
int x = -16;                    // binary: 1111111111110000
int y = 3;
int result = (unsigned int)x >> y;  // binary: 0001111111111110
```

If you are careful to avoid sign extension, you can use the right-shift operator >> as a way to divide by powers of 2. For example:

```
int x = 1000;
int y = x >> 3;   // integer division of 1000 by 8, causing y = 125.
```

++ [Compound Operators]

Description

Increments the value of a variable by 1.

Syntax

```
x++;  // increment x by one and returns the old value of x
++x;  // increment x by one and returns the new value of x
```

Parameters

x: variable. **Allowed data types:** integer, long (possibly unsigned)

Returns

The original or newly incremented value of the variable.

Example Code

```
x = 2;
y = ++x;      // x now contains 3, y contains 3
y = x++;      // x contains 4, but y still contains 3
```

--   [Compound Operators]

Description

Decrements the value of a variable by 1.

Syntax

```
x-- ;   // decrement x by one and returns the old value of x
--x ;   // decrement x by one and returns the new value of x
```

Parameters

`x`: variable. **Allowed data types:** integer, long (possibly unsigned)

Returns

The original or newly decremented value of the variable.

Example Code

```
x = 2;
y = --x;      // x now contains 1, y contains 1
y = x--;      // x contains 0, but y still contains 1
```

+=   [Compound Operators]

Description

This is a convenient shorthand to perform addition on a variable with another constant or variable.

Syntax

```
x += y;   // equivalent to the expression x = x + y;
```

Parameters

`x`: variable. **Allowed data types:** int, float, double, byte, short, long
`y`: variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
x = 2;
x += 4;      // x now contains 6
```

-= [Compound Operators]

Description

This is a convenient shorthand to perform subtraction of a constant or a variable from a variable.

Syntax

```
x -= y;   // equivalent to the expression x = x - y;
```

Parameters

`x`: variable. **Allowed data types:** int, float, double, byte, short, long
`y`: variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
x = 20;
x -= 2;      // x now contains 18
```

*= [Compound Operators]

Description

This is a convenient shorthand to perform multiplication of a variable with another constant or variable.

Syntax

```
x *= y;   // equivalent to the expression x = x * y;
```

Parameters

`x`: variable. **Allowed data types:** int, float, double, byte, short, long
`y`: variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
x = 2;
x *= 2;      // x now contains 4
```

/= [Compound Operators]

Description

This is a convenient shorthand to perform division of a variable with another constant or variable.

Syntax

```
x /= y;   // equivalent to the expression x = x / y;
```

Parameters

`x`: variable. **Allowed data types:** int, float, double, byte, short, long
`y`: **non zero** variable or constant. **Allowed data types:** int, float, double, byte, short, long

Example Code

```
x = 2;
x /= 2;      // x now contains 1
```

&= [Compound Operators]

Description

The compound bitwise AND operator `&=` is often used with a variable and a constant to force particular bits in a variable to the LOW state (to 0). This is often referred to in programming guides as "clearing" or "resetting" bits.

A review of the Bitwise AND `&` operator:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  0  0  1    (operand1 & operand2) - returned result
```

Syntax

```
x &= y;   // equivalent to x = x & y;
```

Parameters

`x`: variable. **Allowed data types:** char, int, long
`y`: variable or constant. **Allowed data types:** char, int, long

Example Code

Bits that are "bitwise ANDed" with 0 are cleared to 0 so, if myByte is a byte variable,

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

Bits that are "bitwise ANDed" with 1 are unchanged so,

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

Notes and Warnings

Because we are dealing with bits in a bitwise operator - it is convenient to use the binary formatter with constants. The numbers are still the same value in other representations, they are just not as easy to understand. Also, B00000000 is shown for clarity, but zero in any number format is zero (hmmm something philosophical there?)

Consequently - to clear (set to zero) bits 0 & 1 of a variable, while leaving the rest of the variable unchanged, use the compound bitwise AND operator (&=) with the constant B11111100

```
1  0  1  0  1  0  1  0    variable
1  1  1  1  1  1  0  0    mask
----------------------
1  0  1  0  1  0  0  0
bits unchanged
                 bits cleared
```

Here is the same representation with the variable's bits replaced with the symbol x

```
x  x  x  x  x  x  x  x    variable
1  1  1  1  1  1  0  0    mask
----------------------
x  x  x  x  x  x  0  0
bits unchanged
                 bits cleared
```

So if:

```
myByte =  B10101010;
myByte &= B11111100; // results in B10101000
```

|= [Compound Operators]

Description

The compound bitwise OR operator `|=` is often used with a variable and a constant to "set" (set to 1) particular bits in a variable.

A review of the Bitwise OR `|` operator:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  1    (operand1 | operand2) - returned result
```

Syntax

```
x |= y;   // equivalent to x = x | y;
```

Parameters

`x`: variable. **Allowed data types:** char, int, long
`y`: variable or constant. **Allowed data types:** char, int, long

Example Code

Bits that are "bitwise ORed" with 0 are unchanged, so if myByte is a byte variable,

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

Bits that are "bitwise ORed" with 1 are set to 1 so:

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

Notes and Warnings

Because we are dealing with bits in a bitwise operator - it is convenient to use the binary formatter with constants. The numbers are still the same value in other representations, they are just not as easy to understand. Also, B00000000 is shown for clarity, but zero in any number format is zero.

Consequently - to set bits 0 & 1 of a variable, while leaving the rest of the variable unchanged, use the compound bitwise OR operator (|=) with the constant B00000011

```
1  0  1  0  1  0  1  0    variable
0  0  0  0  0  0  1  1    mask
----------------------
1  0  1  0  1  0  1  1
bits unchanged
                 bits set
```

Here is the same representation with the variables bits replaced with the symbol x

```
x  x  x  x  x  x  x  x    variable
0  0  0  0  0  0  1  1    mask
----------------------
x  x  x  x  x  x  1  1
bits unchanged
                 bits set
```

So if:

```
myByte =  B10101010;
myByte |= B00000011 == B10101011;
```

Variables

constants

Description

Constants are predefined expressions in the Arduino language. They are used to make the programs easier to read. We classify constants in groups:

Defining Logical Levels: true and false (Boolean Constants)

There are two constants used to represent truth and falsity in the Arduino language: `true`, and `false`.

false

`false` is the easier of the two to define. false is defined as 0 (zero).

true

`true` is often said to be defined as 1, which is correct, but true has a wider definition. Any integer which is non-zero is true, in a Boolean sense. So -1, 2 and -200 are all defined as true, too, in a Boolean sense.

Note that the `true` and `false` constants are typed in lowercase unlike `HIGH`, `LOW`, `INPUT`, and `OUTPUT`.

Defining Pin Levels: HIGH and LOW

When reading or writing to a digital pin there are only two possible values a pin can take/be-set-to: `HIGH` and `LOW`.

HIGH

The meaning of `HIGH` (in reference to a pin) is somewhat different depending on whether a pin is set to an `INPUT` or `OUTPUT`. When a pin is configured as an `INPUT` with pinMode(), and read with digitalRead(), the Arduino (ATmega) will report `HIGH` if:

- a voltage greater than 3.0V is present at the pin (5V boards)
- a voltage greater than 2.0V volts is present at the pin (3.3V boards)

A pin may also be configured as an INPUT with `pinMode()`, and subsequently made HIGH with digitalWrite(). This will enable the internal 20K pullup resistors, which will *pull up* the input pin to a `HIGH` reading unless it is pulled `LOW` by external circuitry. This is how `INPUT_PULLUP` works and is described below in more detail.

When a pin is configured to OUTPUT with `pinMode()`, and set to `HIGH` with `digitalWrite()`, the pin is at:

- 5 volts (5V boards)
- 3.3 volts (3.3V boards)

In this state it can source current, e.g. light an LED that is connected through a series resistor to ground.

LOW

The meaning of `LOW` also has a different meaning depending on whether a pin is set to `INPUT` or `OUTPUT`. When a pin is configured as an `INPUT` with `pinMode()`, and read with `digitalRead()`, the Arduino (ATmega) will report LOW if:

- a voltage less than 1.5V is present at the pin (5V boards)
- a voltage less than 1.0V (Approx) is present at the pin (3.3V boards)

When a pin is configured to `OUTPUT` with `pinMode()`, and set to `LOW` with `digitalWrite()`, the pin is at 0 volts (both 5V and 3.3V boards). In this state it can sink current, e.g. light an LED that is connected through a series resistor to +5 volts (or +3.3 volts).

Defining Digital Pins modes: INPUT, INPUT_PULLUP, and OUTPUT

Digital pins can be used as `INPUT`, `INPUT_PULLUP`, or `OUTPUT`. Changing a pin with `pinMode()` changes the electrical behavior of the pin.

Pins Configured as INPUT

Arduino (ATmega) pins configured as `INPUT` with `pinMode()` are said to be in a *high-impedance* state. Pins configured as `INPUT` make extremely small demands on the circuit that they are sampling, equivalent to a series resistor of 100 Megohms in front of the pin. This makes them useful for reading a sensor.

If you have your pin configured as an `INPUT`, and are reading a switch, when the switch is in the open state the input pin will be "floating", resulting in unpredictable results. In order to assure a proper reading when the switch is open, a pull-up or pull-down resistor must be used. The purpose of this resistor is to pull the pin to a known state when the switch is open. A 10 K ohm resistor is usually chosen, as it is a low enough value to reliably prevent a floating input, and at the same time a high enough value to not not draw too much current when the switch is closed. See the Digital Read Serial tutorial for more information.

If a pull-down resistor is used, the input pin will be `LOW` when the switch is open and `HIGH` when the switch is closed.

If a pull-up resistor is used, the input pin will be `HIGH` when the switch is open and `LOW` when the switch is closed.

Pins Configured as INPUT_PULLUP

The ATmega microcontroller on the Arduino has internal pull-up resistors (resistors that connect to power internally) that you can access. If you prefer to use these instead of external pull-up resistors, you can use the `INPUT_PULLUP` argument in `pinMode()`.

See the Input Pullup Serial tutorial for an example of this in use.

Pins configured as inputs with either `INPUT` or `INPUT_PULLUP` can be damaged or destroyed if they are connected to voltages below ground (negative voltages) or above the positive power rail (5V or 3V).

Pins Configured as OUTPUT

Pins configured as `OUTPUT` with `pinMode()` are said to be in a *low-impedance* state. This means that they can provide a substantial amount of current to other circuits. ATmega pins can source (provide current) or sink (absorb current) up to 40 mA (milliamps) of current to other devices/circuits. This makes them useful for powering LEDs because LEDs typically use less than 40 mA. Loads greater than 40 mA (e.g. motors) will require a transistor or other interface circuitry.

Pins configured as outputs can be damaged or destroyed if they are connected to either the ground or positive power rails.

Defining built-ins: LED_BUILTIN

Most Arduino boards have a pin connected to an on-board LED in series with a resistor. The constant `LED_BUILTIN` is the number of the pin to which the on-board LED is connected. Most boards have this LED connected to digital pin 13.

Integer Constants

Description

Integer constants are numbers that are used directly in a sketch, like 123. By default, these numbers are treated as int but you can change this with the U and L modifiers (see below).

Normally, integer constants are treated as base 10 (decimal) integers, but special notation (formatters) may be used to enter numbers in other bases.

Base	Example	Formatter	Comment
10 (decimal)	123	none	
2 (binary)	B1111011	leading 'B'	only works with 8 bit values (0 to 255) characters 0&1 valid
8 (octal)	0173	leading "0"	characters 0-7 valid
16 (hexadecimal)	0x7B	leading "0x"	characters 0-9, A-F, a-f valid

Decimal (base 10)

This is the common-sense math with which you are acquainted. Constants without other prefixes are assumed to be in decimal format.

Example Code:

```
n = 101;      // same as 101 decimal   ((1 * 10^2) + (0 * 10^1) + 1)
```

Binary (base 2)

Only the characters 0 and 1 are valid.

Example Code:

```
n = B101;    // same as 5 decimal   ((1 * 2^2) + (0 * 2^1) + 1)
```

The binary formatter only works on bytes (8 bits) between 0 (B0) and 255 (B11111111). If it is convenient to input an int (16 bits) in binary form you can do it a two-step procedure such as:

```
myInt = (B11001100 * 256) + B10101010;    // B11001100 is the high byte`
```

Octal (base 8)

Only the characters 0 through 7 are valid. Octal values are indicated by the prefix "0" (zero).

Example Code:

```
n = 0101;    // same as 65 decimal   ((1 * 8^2) + (0 * 8^1) + 1)
```

It is possible to generate a hard-to-find bug by (unintentionally) including a leading zero before a constant and having the compiler unintentionally interpret your constant as octal.

Hexadecimal (base 16)

Valid characters are 0 through 9 and letters A through F; A has the value 10, B is 11, up to F, which is 15. Hex values are indicated by the prefix "0x". Note that A-F may be syted in upper or lower case (a-f).

Example Code:

```
n = 0x101;   // same as 257 decimal   ((1 * 16^2) + (0 * 16^1) + 1)
```

Notes and Warnings

U & L formatters:

By default, an integer constant is treated as an int with the attendant limitations in values. To specify an integer constant with another data type, follow it with:

- a 'u' or 'U' to force the constant into an unsigned data format. Example: 33u
- a 'l' or 'L' to force the constant into a long data format. Example: 100000L
- a 'ul' or 'UL' to force the constant into an unsigned long constant. Example: 32767ul

Floating Point Constants

Description

Similar to integer constants, floating point constants are used to make code more readable. Floating point constants are swapped at compile time for the value to which the expression evaluates.

Example Code

```
n = 0.005;  // 0.005 is a floating point constant
```

Notes and Warnings

Floating point constants can also be expressed in a variety of scientific notation. 'E' and 'e' are both accepted as valid exponent indicators.

floating-point constant	**evaluates to:**	**also evaluates to:**
10.0	10	
2.34E5	2.34 * 10^5	234000
67e-12	67.0 * 10^-12	0.000000000067

Data Types

void [Data Types]

Description

The `void` keyword is used only in function declarations. It indicates that the function is expected to return no information to the function from which it was called.

Example Code

The code shows how to use `void`.

```
// actions are performed in the functions "setup" and "loop"
// but  no information is reported to the larger program

void setup()
{
  // ...
}

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

boolean **[Data Types]**

Description

`boolean` is a non-standard type alias for bool defined by Arduino. It's recommended to instead use the standard type `bool`, which is identical.

bool [Data Types]

Description

A `bool` holds one of two values, `true` or `false`. (Each `bool` variable occupies one byte of memory.)

Example Code

This code shows how to use the `bool` datatype.

```
int LEDpin = 5;       // LED on pin 5
int switchPin = 13;   // momentary switch on 13, other side connected to ground

bool running = false;

void setup()
{
  pinMode(LEDpin, OUTPUT);
  pinMode(switchPin, INPUT);
  digitalWrite(switchPin, HIGH);      // turn on pullup resistor
}

void loop()
{
  if (digitalRead(switchPin) == LOW)
  {  // switch is pressed - pullup keeps pin high normally
    delay(100);                        // delay to debounce switch
    running = !running;                // toggle running variable
    digitalWrite(LEDpin, running);     // indicate via LED
  }
}
```

char **[Data Types]**

Description

A data type that takes up 1 byte of memory that stores a character value. Character literals are written in single quotes, like this: 'A' (for multiple characters - strings - use double quotes: "ABC").

Characters are stored as numbers however. You can see the specific encoding in the ASCII chart. This means that it is possible to do arithmetic on characters, in which the ASCII value of the character is used (e.g. 'A' + 1 has the value 66, since the ASCII value of the capital letter A is 65). See `Serial.println` reference for more on how characters are translated to numbers.

The char datatype is a signed type, meaning that it encodes numbers from -128 to 127. For an unsigned, one-byte (8 bit) data type, use the *byte* data type.

Example Code

```
  char myChar = 'A';
  char myChar = 65;      // both are equivalent
```

unsigned char **[Data Types]**

Description

An unsigned data type that occupies 1 byte of memory. Same as the byte datatype.

The unsigned char datatype encodes numbers from 0 to 255.

For consistency of Arduino programming style, the byte data type is to be preferred.

Example Code

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

byte [Data Types]

Description

A byte stores an 8-bit unsigned number, from 0 to 255.

Example Code

int **[Data Types]**

Description

Integers are your primary data-type for number storage.

On the Arduino Uno (and other ATmega based boards) an int stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^15 and a maximum value of (2^15) - 1). On the Arduino Due and SAMD based boards (like MKR1000 and Zero), an int stores a 32-bit (4-byte) value. This yields a range of -2,147,483,648 to 2,147,483,647 (minimum value of -2^31 and a maximum value of (2^31) - 1).

int’s store negative numbers with a technique called (2’s complement math). The highest bit, sometimes referred to as the "sign" bit, flags the number as a negative number. The rest of the bits are inverted and 1 is added.

The Arduino takes care of dealing with negative numbers for you, so that arithmetic operations work transparently in the expected manner. There can be an unexpected complication in dealing with the bitshift right operator (>>) however.

Syntax

```
int var = val;
```

`var` - your int variable name
`val` - the value you assign to that variable

Example Code

```
  int ledPin = 13;
```

Notes and Warnings

When signed variables are made to exceed their maximum or minimum capacity they *overflow*. The result of an overflow is unpredictable so this should be avoided. A typical symptom of an overflow is the variable "rolling over" from its maximum capacity to its minimum or vice versa, but this is not always the case. If you want this behavior, use unsigned int.

unsigned int [Data Types]

Description

On the Uno and other ATMEGA based boards, unsigned ints (unsigned integers) are the same as ints in that they store a 2 byte value. Instead of storing negative numbers however they only store positive values, yielding a useful range of 0 to 65,535 ((2^16) - 1).

The Due stores a 4 byte (32-bit) value, ranging from 0 to 4,294,967,295 (2^32 - 1).

The difference between unsigned ints and (signed) ints, lies in the way the highest bit, sometimes referred to as the "sign" bit, is interpreted. In the Arduino int type (which is signed), if the high bit is a "1", the number is interpreted as a negative number, and the other 15 bits are interpreted with (2's complement math).

Syntax

`unsigned int var = val;` `var` - your unsigned int variable name `val` - the value you assign to that variable

Example Code

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

Notes and Warnings

When unsigned variables are made to exceed their maximum capacity they "roll over" back to 0, and also the other way around:

```
unsigned int x;
   x = 0;
   x = x - 1;       // x now contains 65535 - rolls over in neg direction
   x = x + 1;       // x now contains 0 - rolls over
```

Math with unsigned variables may produce unexpected results, even if your unsigned variable never rolls over.

The MCU applies the following rules:

The calculation is done in the scope of the destination variable. E.g. if the destination variable is signed, it will do signed math, even if both input variables are unsigned.

However with a calculation which requires an intermediate result, the scope of the intermediate result is unspecified by the code. In this case, the MCU will do unsigned math for the intermediate result, because both inputs are unsigned!

```
unsigned int x=5;
unsigned int y=10;
int result;

   result = x - y; // 5 - 10 = -5, as expected
   result = (x - y)/2; // 5 - 10 in unsigned math is 65530!  65530/2 = 32765

   // solution: use signed variables, or do the calculation step by step.
   result = x - y; // 5 - 10 = -5, as expected
   result = result / 2; //  -5/2 = -2 (only integer math, decimal places are
dropped)
```

Why use unsigned variables at all?

- The rollover behaviour is desired, e.g. counters
- The signed variable is a bit too small, but you want to avoid the memory and speed loss of long/float.

word [Data Types]

Description

A word stores a 16-bit unsigned number, from 0 to 65535. Same as an unsigned int.

Example Code

```
  word w = 10000;
```

long **[Data Types]**

Description

Long variables are extended size variables for number storage, and store 32 bits (4 bytes), from -2,147,483,648 to 2,147,483,647.

If doing math with integers, at least one of the numbers must be followed by an L, forcing it to be a long. See the Integer Constants page for details.

Syntax

```
long var = val;
```

`var` - the long variable name `val` - the value assigned to the variable

Example Code

```
  long speedOfLight = 186000L;   // see the Integer Constants page for
explanation of the 'L'
```

unsigned long [Data Types]

Description

Unsigned long variables are extended size variables for number storage, and store 32 bits (4 bytes). Unlike standard longs unsigned longs won’t store negative numbers, making their range from 0 to 4,294,967,295 (2^32 - 1).

Syntax

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

`var` - your long variable name `val` - the value you assign to that variable

Example Code

```
unsigned long time;

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

void loop()
{
  Serial.print("Time: ");
  time = millis();
  //prints time since program started
  Serial.println(time);
  // wait a second so as not to send massive amounts of data
  delay(1000);
}
```

short [Data Types]

Description

A short is a 16-bit data-type.

On all Arduinos (ATMega and ARM based) a short stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^15 and a maximum value of (2^15) - 1).

Syntax

```
short var = val;
```

`var` - your short variable name `val` - the value you assign to that variable

Example Code

```
 short ledPin = 13
```

float [Data Types]

Description

Datatype for floating-point numbers, a number that has a decimal point. Floating-point numbers are often used to approximate analog and continuous values because they have greater resolution than integers. Floating-point numbers can be as large as 3.4028235E+38 and as low as -3.4028235E+38. They are stored as 32 bits (4 bytes) of information.

Floats have only 6-7 decimal digits of precision. That means the total number of digits, not the number to the right of the decimal point. Unlike other platforms, where you can get more precision by using a double (e.g. up to 15 digits), on the Arduino, double is the same size as float.

Floating point numbers are not exact, and may yield strange results when compared. For example 6.0 / 3.0 may not equal 2.0. You should instead check that the absolute value of the difference between the numbers is less than some small number.

Floating point math is also much slower than integer math in performing calculations, so should be avoided if, for example, a loop has to run at top speed for a critical timing function. Programmers often go to some lengths to convert floating point calculations to integer math to increase speed.

If doing math with floats, you need to add a decimal point, otherwise it will be treated as an int. See the Floating point constants page for details.

Syntax

```
float var=val;
```

`var` - your float variable name `val` - the value you assign to that variable

Example Code

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

```
int x;
int y;
float z;

x = 1;
y = x / 2;            // y now contains 0, ints can't hold fractions
z = (float)x / 2.0;   // z now contains .5 (you have to use 2.0, not 2)
```

double [Data Types]

Description

Double precision floating point number. On the Uno and other ATMEGA based boards, this occupies 4 bytes. That is, the double implementation is exactly the same as the float, with no gain in precision.

On the Arduino Due, doubles have 8-byte (64 bit) precision.

Notes and Warnings

Users who borrow code from other sources that includes double variables may wish to examine the code to see if the implied precision is different from that actually achieved on ATMEGA based Arduinos.

String **[Data Types]**

Description

Text Strings can be represented in two ways. you can use the String data type, which is part of the core as of version 0019, or you can make a String out of an array of type char and null-terminate it. This page described the latter method. For more details on the String object, which gives you more functionality at the cost of more memory, see the String object page.

Syntax

All of the following are valid declarations for Strings.

```
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";
```

Possibilities for declaring Strings

- Declare an array of chars without initializing it as in Str1
- Declare an array of chars (with one extra char) and the compiler will add the required null character, as in Str2
- Explicitly add the null character, Str3

- Initialize with a String constant in quotation marks; the compiler will size the array to fit the String constant and a terminating null character, Str4
- Initialize the array with an explicit size and String constant, Str5
- Initialize the array, leaving extra space for a larger String, Str6

Null termination

Generally, Strings are terminated with a null character (ASCII code 0). This allows functions (like `Serial.print()`) to tell where the end of a String is. Otherwise, they would continue reading subsequent bytes of memory that aren't actually part of the String.

This means that your String needs to have space for one more character than the text you want it to contain. That is why Str2 and Str5 need to be eight characters, even though "arduino" is only seven - the last position is automatically filled with a null character. Str4 will be automatically sized to eight characters, one for the extra null. In Str3, we've explicitly included the null character (written '\0') ourselves.

Note that it's possible to have a String without a final null character (e.g. if you had specified the length of Str2 as seven instead of eight). This will break most functions that use Strings, so you shouldn't do it intentionally. If you notice something behaving strangely (operating on characters not in the String), however, this could be the problem.

Single quotes or double quotes?

Strings are always defined inside double quotes ("Abc") and characters are always defined inside single quotes('A').

Wrapping long Strings

You can wrap long Strings like this:

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

Arrays of Strings

It is often convenient, when working with large amounts of text, such as a project with an LCD display, to setup an array of Strings. Because Strings themselves are arrays, this is in actually an example of a two-dimensional array.

In the code below, the asterisk after the datatype `char` "char*" indicates that this is an array of "pointers". All array names are actually pointers, so this is required to make an array of arrays. Pointers are one of the more esoteric parts of C for beginners to understand, but it isn't necessary to understand pointers in detail to use them effectively here.

Example Code

```
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);
   }
}
```

String() [Data Types]

Description

Constructs an instance of the String class. There are multiple versions that construct Strings from different data types (i.e. format them as sequences of characters), including:

- a constant string of characters, in double quotes (i.e. a char array)
- a single constant character, in single quotes
- another instance of the String object
- a constant integer or long integer
- a constant integer or long integer, using a specified base
- an integer or long integer variable
- an integer or long integer variable, using a specified base
- a float or double, using a specified decimal palces

Constructing a String from a number results in a string that contains the ASCII representation of that number. The default is base ten, so

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

gives you the String "13". You can use other bases, however. For example,

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

gives you the String "D", which is the hexadecimal representation of the decimal value 13. Or if you prefer binary,

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

gives you the String "1101", which is the binary representation of 13.

Syntax

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

Parameters

`val`: a variable to format as a String - **Allowed data types:** string, char, byte, int, long, unsigned int, unsigned long, float, double
`base` (optional): the base in which to format an integral value `decimalPlaces` (**only if val is float or double**): the desired decimal places

Returns

an instance of the String class.

Example Code

All of the following are valid declarations for Strings.

```
String stringOne = "Hello String";                                     // using
a constant String
String stringOne =  String('a');                                          //
converting a constant char into a String
String stringTwo =  String("This is a string");                // converting a
constant string into a String object
String stringOne =  String(stringTwo + " with more"); // concatenating two
strings
String stringOne =  String(13);                                          //
using a constant integer
String stringOne =  String(analogRead(0), DEC);          // using an int and a
base
String stringOne =  String(45, HEX);                                // using an
int and a base (hexadecimal)
String stringOne =  String(255, BIN);                               // using an
int and a base (binary)
String stringOne =  String(millis(), DEC);                       // using a
long and a base
String stringOne =  String(5.698, 3);                                // using a
float and the decimal places
```

array [Data Types]

Description

An array is a collection of variables that are accessed with an index number. Arrays in the C programming language, on which Arduino is based, can be complicated, but using simple arrays is relatively straightforward.

Creating (Declaring) an Array

All of the methods below are valid ways to create (declare) an array.

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

You can declare an array without initializing it as in myInts.
In myPins we declare an array without explicitly choosing a size. The compiler counts the elements and creates an array of the appropriate size.
Finally you can both initialize and size your array, as in mySensVals. Note that when declaring an array of type char, one more element than your initialization is required, to hold the required null character.

Accessing an Array

Arrays are zero indexed, that is, referring to the array initialization above, the first element of the array is at index 0, hence

`mySensVals[0] == 2, mySensVals[1] == 4,` and so forth.

It also means that in an array with ten elements, index nine is the last element. Hence:

```
int myArray[10]={9,3,2,4,3,2,7,8,9,11};
     // myArray[9]    contains 11
     // myArray[10]   is invalid and contains random information (other memory
address)
```

For this reason you should be careful in accessing arrays. Accessing past the end of an array (using an index number greater than your declared array size - 1) is reading from memory that is in use for other purposes. Reading from these locations is probably not going to do much except yield invalid data. Writing to random memory locations is definitely a bad idea and can often lead to unhappy results such as crashes or program malfunction. This can also be a difficult bug to track down.

Unlike BASIC or JAVA, the C compiler does no checking to see if array access is within legal bounds of the array size that you have declared.

To assign a value to an array:

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

To retrieve a value from an array:

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

Arrays and FOR Loops

Arrays are often manipulated inside for loops, where the loop counter is used as the index for each array element. For example, to print the elements of an array over the serial port, you could do something like this:

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

Example Code

For a complete program that demonstrates the use of arrays, see the (Knight Rider example) from the (Tutorials).

Conversion

char() [Conversion]

Description

Converts a value to the char data type.

Syntax

```
char(x)
```

Parameters

`x`: a value of any type

Returns

```
char
```

byte() [Conversion]

Description

Converts a value to the byte data type.

Syntax

```
byte(x)
```

Parameters

`x`: a value of any type

Returns

```
byte
```

int() [Conversion]

Description

Converts a value to the int data type.

Syntax

```
int(x)
```

Parameters

`x`: a value of any type

Returns

`int`

word() [Conversion]

Description

Converts a value to the word data type.

Syntax

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

Parameters

`x`: a value of any type

`h`: the high-order (leftmost) byte of the word

`l`: the low-order (rightmost) byte of the word

Returns

`Word`

long() [Conversion]

Description

Converts a value to the long data type.

Syntax

```
long(x)
```

Parameters

`x`: a value of any type

Returns

`long`

float() [Conversion]

Description

Converts a value to the float data type.

Syntax

```
float(x)
```

Parameters

`x`: a value of any type

Returns

```
float
```

Notes and Warnings

See the reference for float for details about the precision and limitations of floating point numbers on Arduino.

Variable Scope & Qualifiers

scope [Variable Scope & Qualifiers]

Description

Variables in the C programming language, which Arduino uses, have a property called scope. This is in contrast to early versions of languages such as BASIC where every variable is a *global* variable.

A global variable is one that can be seen by every function in a program. Local variables are only visible to the function in which they are declared. In the Arduino environment, any variable declared outside of a function (e.g. setup(), loop(), etc.), is a *global* variable.

When programs start to get larger and more complex, local variables are a useful way to insure that only one function has access to its own variables. This prevents programming errors when one function inadvertently modifies variables used by another function.

It is also sometimes handy to declare and initialize a variable inside a `for` loop. This creates a variable that can only be accessed from inside the for-loop brackets.

Example Code

```
int gPWMval;  // any function will see this variable

void setup()
{
  // ...
}

void loop()
{
  int i;    // "i" is only "visible" inside of "loop"
  float f;  // "f" is only "visible" inside of "loop"
  // ...

  for (int j = 0; j <100; j++){
  // variable j can only be accessed inside the for-loop brackets
  }

}
```

static [Variable Scope & Qualifiers]

Description

The `static` keyword is used to create variables that are visible to only one function. However unlike local variables that get created and destroyed every time a function is called, static variables persist beyond the function call, preserving their data between function calls.

Variables declared as static will only be created and initialized the first time a function is called.

Example Code

```
/* RandomWalk
* Paul Badger 2007
* RandomWalk wanders up and down randomly between two
* endpoints. The maximum move in one loop is governed by
* the parameter "stepsize".
* A static variable is moved up and down a random amount.
* This technique is also known as "pink noise" and "drunken walk".
*/

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

int thisTime;
int total;

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

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

int randomWalk(int moveSize){
  static int  place;     // variable to store value in random walk - declared
static so that it stores
```

```
                          // values in between function calls, but no other
functions can change its value

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

  if (place < randomWalkLowRange){                              // check lower
and upper limits
    place = randomWalkLowRange + (randomWalkLowRange - place);  // reflect
number back in positive direction
  }
  else if(place > randomWalkHighRange){
    place = randomWalkHighRange - (place - randomWalkHighRange);  // reflect
number back in negative direction
  }

  return place;
}
```

volatile [Variable Scope & Qualifiers]

Description

`volatile` is a keyword known as a variable *qualifier*, it is usually used before the datatype of a variable, to modify the way in which the compiler and subsequent program treats the variable.

Declaring a variable volatile is a directive to the compiler. The compiler is software which translates your C/C++ code into the machine code, which are the real instructions for the Atmega chip in the Arduino.

Specifically, it directs the compiler to load the variable from RAM and not from a storage register, which is a temporary memory location where program variables are stored and manipulated. Under certain conditions, the value for a variable stored in registers can be inaccurate.

A variable should be declared volatile whenever its value can be changed by something beyond the control of the code section in which it appears, such as a concurrently executing thread. In the Arduino, the only place that this is likely to occur is in sections of code associated with interrupts, called an interrupt service routine.

int or long volatiles

If the volatile variable is bigger than a byte (e.g. a 16 bit int or a 32 bit long), then the microcontroller can not read it in one step, because it is an 8 bit microcontroller. This means that while your main code section (e.g. your loop) reads the first 8 bits of the variable, the interrupt might already change the second 8 bits. This will produce random values for the variable.

Remedy:

While the variable is read, interrupts need to be disabled, so they can't mess with the bits, while they are read. There are several ways to do this:

1. LANGUAGE noInterrupts
2. use the ATOMIC_BLOCK macro. Atomic operations are single MCU operations - the smallest possible unit.

Example Code

```
// toggles LED when interrupt pin changes state

int pin = 13;
volatile byte state = LOW;

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

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

void blink()
{
  state = !state;
}
#include <util/atomic.h> // this library includes the ATOMIC_BLOCK macro.
volatile int input_from_interrupt;

  ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
     // code with interrupts blocked (consecutive atomic operations will not get
interrupted)
     int result = input_from_interrupt;
    }
```

const [Variable Scope & Qualifiers]

Description

The `const` keyword stands for constant. It is a variable *qualifier* that modifies the behavior of the variable, making a variable "*read-only*". This means that the variable can be used just as any other variable of its type, but its value cannot be changed. You will get a compiler error if you try to assign a value to a `const` variable.

Constants defined with the `const` keyword obey the rules of variable scoping that govern other variables. This, and the pitfalls of using `#define`, makes the `const` keyword a superior method for defining constants and is preferred over using `#define`.

Example Code

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

// ....

x = pi * 2;    // it's fine to use consts in math

pi = 7;        // illegal - you can't write to (modify) a constant
```

Notes and Warnings

`#define` or `const`

You can use either `const` or `#define` for creating numeric or string constants. For arrays, you will need to use `const`. In general `const` is preferred over `#define` for defining constants.

Utilities

sizeof() [Utilities]

Description

The sizeof operator returns the number of bytes in a variable type, or the number of bytes occupied by an array.

Syntax

`sizeof(variable)`

Parameters

`variable`: any variable type or array (e.g. int, float, byte)

Returns

The number of bytes in a variable or bytes occupied in an array. (size_t)

Example Code

The `sizeof` operator is useful for dealing with arrays (such as strings) where it is convenient to be able to change the size of the array without breaking other parts of the program.

This program prints out a text string one character at a time. Try changing the text phrase.

```
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); // slow down the program
}
```

Notes and Warnings

Note that `sizeof` returns the total number of bytes. So for larger variable types such as ints, the for loop would look something like this. Note also that a properly formatted string ends with the NULL symbol, which has ASCII value 0.

```
for (i = 0; i < (sizeof(myInts)/sizeof(int)); i++) {
  // do something with myInts[i]
}
```

PROGMEM **[Utilities]**

Description

Store data in flash (program) memory instead of SRAM. There's a description of the various types of memory available on an Arduino board.

The `PROGMEM` keyword is a variable modifier, it should be used only with the datatypes defined in pgmspace.h. It tells the compiler "put this information into flash memory", instead of into SRAM, where it would normally go.

PROGMEM is part of the pgmspace.h library. It is included automatically in modern versions of the IDE, however if you are using an IDE version below 1.0 (2011), you'll first need to include the library at the top your sketch, like this:

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

Syntax

const dataType variableName[] PROGMEM = {data0, data1, data3…};

`dataType` - any variable type `variableName` - the name for your array of data

Note that because PROGMEM is a variable modifier, there is no hard and fast rule about where it should go, so the Arduino compiler accepts all of the definitions below, which are also synonymous. However experiments have indicated that, in various versions of Arduino (having to do with GCC version), PROGMEM may work in one location and not in another. The "string table" example below has been tested to work with Arduino 13. Earlier versions of the IDE may work better if PROGMEM is included after the variable name.

`const dataType variableName[] PROGMEM = {}; // use this form`
`const PROGMEM dataType variableName[] = {}; // or this one`+ `const dataType PROGMEM variableName[] = {}; // not this one

While `PROGMEM` could be used on a single variable, it is really only worth the fuss if you have a larger block of data that needs to be stored, which is usually easiest in an array, (or another C data structure beyond our present discussion).

Using `PROGMEM` is also a two-step procedure. After getting the data into Flash memory, it requires special methods (functions), also defined in the http://www.nongnu.org/avr-libc/user-manual/group*avr*pgmspace.html[pgmspace.h] library, to read the data from program memory back into SRAM, so we can do something useful with it.

Example Code

The following code fragments illustrate how to read and write unsigned chars (bytes) and ints (2 bytes) to PROGMEM.

```
// save some unsigned ints
const PROGMEM  uint16_t charSet[]  = { 65000, 32796, 16843, 10, 11234};

// save some chars
const char signMessage[] PROGMEM  = {"I AM PREDATOR,  UNSEEN COMBATANT. CREATED
BY THE UNITED STATES DEPART"};

unsigned int displayInt;
int k;    // counter variable
char myChar;


void setup() {
  Serial.begin(9600);
  while (!Serial);  // wait for serial port to connect. Needed for native USB

  // put your setup code here, to run once:
  // read back a 2-byte int
  for (k = 0; k < 5; k++)
  {
    displayInt = pgm_read_word_near(charSet + k);
    Serial.println(displayInt);
  }
  Serial.println();

  // read back a char
  for (k = 0; k < strlen_P(signMessage); k++)
  {
    myChar =  pgm_read_byte_near(signMessage + k);
    Serial.print(myChar);
  }

  Serial.println();
}

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

}
```

Arrays of strings

It is often convenient when working with large amounts of text, such as a project with an LCD display, to setup an array of strings. Because strings themselves are arrays, this is in actually an example of a two-dimensional array.

These tend to be large structures so putting them into program memory is often desirable. The code below illustrates the idea.

```
/*
 PROGMEM string demo
 How to store a table of strings in program memory (flash),
 and retrieve them.

 Information summarized from:
 http://www.nongnu.org/avr-libc/user-manual/pgmspace.html

 Setting up a table (array) of strings in program memory is slightly
complicated, but
 here is a good template to follow.
 Setting up the strings is a two-step process. First define the strings.
*/

#include <avr/pgmspace.h>
const char string_0[] PROGMEM = "String 0";   // "String 0" etc are strings to
store - change to suit.
const char string_1[] PROGMEM = "String 1";
const char string_2[] PROGMEM = "String 2";
const char string_3[] PROGMEM = "String 3";
const char string_4[] PROGMEM = "String 4";
const char string_5[] PROGMEM = "String 5";


// Then set up a table to refer to your strings.

const char* const string_table[] PROGMEM = {string_0, string_1, string_2,
string_3, string_4, string_5};

char buffer[30];    // make sure this is large enough for the largest string it
must hold

void setup()
{
  Serial.begin(9600);
  while(!Serial); // wait for serial port to connect. Needed for native USB
  Serial.println("OK");
}


void loop()
{
  /* Using the string table in program memory requires the use of special
functions to retrieve the data.
     The strcpy_P function copies a string from program space to a string in RAM
("buffer").
     Make sure your receiving string in RAM  is large enough to hold whatever
     you are retrieving from program space. */


  for (int i = 0; i < 6; i++)
  {
    strcpy_P(buffer, (char*)pgm_read_word(&(string_table[i]))); // Necessary
casts and dereferencing, just copy.
    Serial.println(buffer);
    delay( 500 );
  }
}
```

Notes and Warnings

Please note that variables must be either globally defined, OR defined with the static keyword, in order to work with PROGMEM.

The following code will NOT work when inside a function:

```
const char long_str[] PROGMEM = "Hi, I would like to tell you a bit about
myself.\n";
```

The following code WILL work, even if locally defined within a function:

```
const static char long_str[] PROGMEM = "Hi, I would like to tell you a bit about
myself.\n"
```

=== The `F()` macro

When an instruction like :

```
Serial.print("Write something on  the Serial Monitor");
```

is used, the string to be printed is normally saved in RAM. If your sketch prints a lot of stuff on the Serial Monitor, you can easily fill the RAM. If you have free FLASH memory space, you can easily indicate that the string must be saved in FLASH using the syntax:

```
Serial.print(F("Write something on the Serial Monitor that is stored in
FLASH"));
```

Functions

Digital I/O

pinMode() [Digital I/O]

Description

Configures the specified pin to behave either as an input or an output. See the description of (digital pins) for details on the functionality of the pins.

As of Arduino 1.0.1, it is possible to enable the internal pullup resistors with the mode INPUT_PULLUP. Additionally, the INPUT mode explicitly disables the internal pullups.

Syntax

```
pinMode(pin, mode)
```

Parameters

`pin`: the number of the pin whose mode you wish to set

`mode`: `INPUT`, `OUTPUT`, or `INPUT_PULLUP`. (see the (digital pins) page for a more complete description of the functionality.)

Returns

Nothing

Example Code

The code makes the digital pin 13 `OUTPUT` and Toggles it `HIGH` and `LOW`

```
void setup()
{
  pinMode(13, OUTPUT);          // sets the digital pin 13 as output
}

void loop()
{
  digitalWrite(13, HIGH);       // sets the digital pin 13 on
  delay(1000);                  // waits for a second
  digitalWrite(13, LOW);        // sets the digital pin 13 off
  delay(1000);                  // waits for a second
}
```

Notes and Warnings

The analog input pins can be used as digital pins, referred to as A0, A1, etc.

digitalWrite() [Digital I/O]

Description

Write a `HIGH` or a `LOW` value to a digital pin.

If the pin has been configured as an `OUTPUT` with `pinMode()`, its voltage will be set to the corresponding value: 5V (or 3.3V on 3.3V boards) for `HIGH`, 0V (ground) for `LOW`.

If the pin is configured as an `INPUT`, `digitalWrite()` will enable (`HIGH`) or disable (`LOW`) the internal pullup on the input pin. It is recommended to set the `pinMode()` to `INPUT_PULLUP` to enable the internal pull-up resistor. See the digital pins tutorial for more information.

If you do not set the `pinMode()` to `OUTPUT`, and connect an LED to a pin, when calling `digitalWrite(HIGH)`, the LED may appear dim. Without explicitly setting `pinMode()`, `digitalWrite()` will have enabled the internal pull-up resistor, which acts like a large current-limiting resistor.

Syntax

`digitalWrite(pin, value)`

Parameters

`pin`: the pin number

`value`: `HIGH` or `LOW`

Returns

Nothing

Example Code

The code makes the digital pin 13 an `OUTPUT` and toggles it by alternating between `HIGH` and `LOW` at one second pace.

```
void setup()
{
  pinMode(13, OUTPUT);          // sets the digital pin 13 as output
}

void loop()
{
  digitalWrite(13, HIGH);       // sets the digital pin 13 on
  delay(1000);                  // waits for a second
  digitalWrite(13, LOW);        // sets the digital pin 13 off
  delay(1000);                  // waits for a second
}
```

Notes and Warnings

The analog input pins can be used as digital pins, referred to as A0, A1, etc.

digitalRead() [Digital I/O]

Description

Reads the value from a specified digital pin, either `HIGH` or `LOW`.

Syntax

`digitalRead(pin)`

Parameters

`pin`: the number of the digital pin you want to read

Returns

`HIGH` or `LOW`

Example Code

Sets pin 13 to the same value as pin 7, declared as an input.

```
int ledPin = 13;   // LED connected to digital pin 13
int inPin = 7;     // pushbutton connected to digital pin 7
int val = 0;       // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin 13 as output
  pinMode(inPin, INPUT);        // sets the digital pin 7 as input
}

void loop()
{
  val = digitalRead(inPin);     // read the input pin
  digitalWrite(ledPin, val);    // sets the LED to the button's value
}
```

Notes and Warnings

If the pin isn't connected to anything, digitalRead() can return either HIGH or LOW (and this can change randomly).

The analog input pins can be used as digital pins, referred to as A0, A1, etc.

Analog I/O

analogReference() [Analog I/O]

Description

Configures the reference voltage used for analog input (i.e. the value used as the top of the input range). The options are:

Arduino AVR Boards (Uno, Mega, etc.)

- DEFAULT: the default analog reference of 5 volts (on 5V Arduino boards) or 3.3 volts (on 3.3V Arduino boards)
- INTERNAL: an built-in reference, equal to 1.1 volts on the ATmega168 or ATmega328P and 2.56 volts on the ATmega8 (not available on the Arduino Mega)
- INTERNAL1V1: a built-in 1.1V reference (Arduino Mega only)
- INTERNAL2V56: a built-in 2.56V reference (Arduino Mega only)
- EXTERNAL: the voltage applied to the AREF pin (0 to 5V only) is used as the reference.

Arduino SAMD Boards (Zero, etc.)

- AR_DEFAULT: the default analog reference of 3.3V
- AR_INTERNAL: a built-in 2.23V reference
- AR_INTERNAL1V0: a built-in 1.0V reference
- AR_INTERNAL1V65: a built-in 1.65V reference
- AR_INTERNAL2V23: a built-in 2.23V reference
- AR_EXTERNAL: the voltage applied to the AREF pin is used as the reference

Arduino SAM Boards (Due)

- AR_DEFAULT: the default analog reference of 3.3V. This is the only supported option for the Due.

Syntax

```
analogReference(type)
```

Parameters

`type`: which type of reference to use (see list of options in the description).

Returns

Nothing

Notes and Warnings

After changing the analog reference, the first few readings from `analogRead()` may not be accurate.

Don't use anything less than 0V or more than 5V for external reference voltage on the AREF pin! If you're using an external reference on the AREF pin, you must set the analog reference to EXTERNAL before calling `analogRead()`. Otherwise, you will short together the active reference voltage (internally generated) and the AREF pin, possibly damaging the microcontroller on your Arduino board.

Alternatively, you can connect the external reference voltage to the AREF pin through a 5K resistor, allowing you to switch between external and internal reference voltages. Note that the resistor will alter the voltage that gets used as the reference because there is an internal 32K resistor on the AREF pin. The two act as a voltage divider, so, for example, 2.5V applied through the resistor will yield 2.5 * 32 / (32 + 5) = ~2.2V at the AREF pin.

analogRead() [Analog I/O]

Description

Reads the value from the specified analog pin. The Arduino board contains a 6 channel (8 channels on the Mini and Nano, 16 on the Mega), 10-bit analog to digital converter. This means that it will map input voltages between 0 and 5 volts into integer values between 0 and 1023. This yields a resolution between readings of: 5 volts / 1024 units or, .0049 volts (4.9 mV) per unit. The input range and resolution can be changed using analogReference().

It takes about 100 microseconds (0.0001 s) to read an analog input, so the maximum reading rate is about 10,000 times a second.

Syntax

```
analogRead(pin)
```

Parameters

`pin`: the number of the analog input pin to read from (0 to 5 on most boards, 0 to 7 on the Mini and Nano, 0 to 15 on the Mega)

Returns

int(0 to 1023)

Example Code

The code reads the voltage on analogPin and displays it.

```
int analogPin = 3;      // potentiometer wiper (middle terminal) connected to
analog pin 3
                        // outside leads to ground and +5V
int val = 0;            // variable to store the value read

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

void loop()
{
  val = analogRead(analogPin);     // read the input pin
  Serial.println(val);             // debug value
}
```

Notes and Warnings

If the analog input pin is not connected to anything, the value returned by analogRead() will fluctuate based on a number of factors (e.g. the values of the other analog inputs, how close your hand is to the board, etc.).

analogWrite() [Analog I/O]

Description

Writes an analog value (PWM wave) to a pin. Can be used to light a LED at varying brightnesses or drive a motor at various speeds. After a call to `analogWrite()`, the pin will generate a steady square wave of the specified duty cycle until the next call to `analogWrite()` (or a call to `digitalRead()` or `digitalWrite()`) on the same pin. The frequency of the PWM signal on most pins is approximately 490 Hz. On the Uno and similar boards, pins 5 and 6 have a frequency of approximately 980 Hz.

On most Arduino boards (those with the ATmega168 or ATmega328P), this function works on pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega, it works on pins 2 - 13 and 44 - 46. Older Arduino boards with an ATmega8 only support `analogWrite()` on pins 9, 10, and 11.
The Arduino DUE supports `analogWrite()` on pins 2 through 13, plus pins DAC0 and DAC1. Unlike the PWM pins, DAC0 and DAC1 are Digital to Analog converters, and act as true analog outputs.
You do not need to call `pinMode()` to set the pin as an output before calling `analogWrite()`.
The `analogWrite` function has nothing to do with the analog pins or the `analogRead` function.

Syntax

`analogWrite(pin, value)`

Parameters

`pin`: the pin to write to. Allowed data types: int.
`value`: the duty cycle: between 0 (always off) and 255 (always on). Allowed data types: int

Returns

Nothing

Example Code

Sets the output to the LED proportional to the value read from the potentiometer.

```
int ledPin = 9;      // LED connected to digital pin 9
int analogPin = 3;   // potentiometer connected to analog pin 3
int val = 0;         // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT);   // sets the pin as output
}

void loop()
{
  val = analogRead(analogPin);   // read the input pin
  analogWrite(ledPin, val / 4);  // analogRead values go from 0 to 1023,
analogWrite values from 0 to 255
}
```

Notes and Warnings

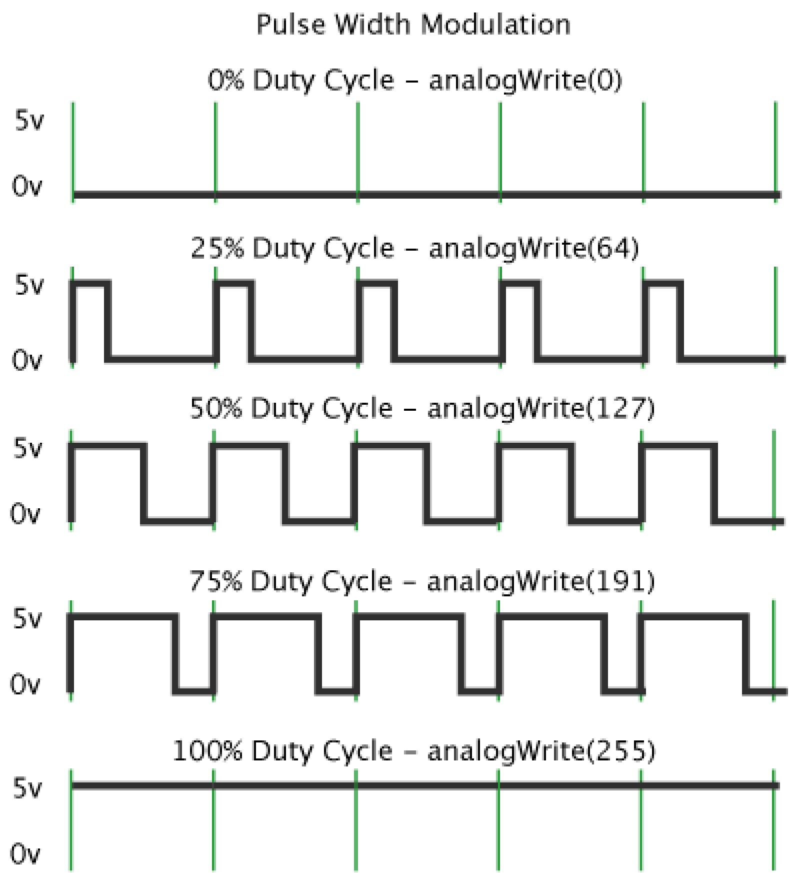
The PWM outputs generated on pins 5 and 6 will have higher-than-expected duty cycles. This is because of interactions with the `millis()` and `delay()` functions, which share the same internal timer used to generate those PWM outputs. This will be noticed mostly on low duty-cycle settings (e.g. 0 - 10) and may result in a value of 0 not fully turning off the output on pins 5 and 6.

PWM

The Fading example demonstrates the use of analog output (PWM) to fade an LED. It is available in the File->Sketchbook->Examples->Analog menu of the Arduino software.

Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off. The duration of "on time" is called the pulse width. To get varying analog values, you change, or modulate, that pulse width. If you repeat this on-off pattern fast enough with an LED for example, the result is as if the signal is a steady voltage between 0 and 5v controlling the brightness of the LED.

In the graphic below, the green lines represent a regular time period. This duration or period is the inverse of the PWM frequency. In other words, with Arduino's PWM frequency at about 500Hz, the green lines would measure 2 milliseconds each. A call to analogWrite() is on a scale of 0 - 255, such that analogWrite(255) requests a 100% duty cycle (always on), and analogWrite(127) is a 50% duty cycle (on half the time) for example.



Pulse Width Modulation
0% Duty Cycle - analogWrite(0)
5v
0v
25% Duty Cycle - analogWrite(64)
5v
0v
50% Duty Cycle - analogWrite(127)
5v
0v
75% Duty Cycle - analogWrite(191)
5v
0v
100% Duty Cycle - analogWrite(255)
5v
0v

Once you get this example running, grab your arduino and shake it back and forth. What you are doing here is essentially mapping time across the space. To our eyes, the movement blurs each LED blink into a line. As the LED fades in and out, those little lines will grow and shrink in length. Now you are seeing the pulse width.

Zero, Due & MKR Family

analogReadResolution() [Zero, Due & MKR Family]

Description

analogReadResolution() is an extension of the Analog API for the Arduino Due, Zero and MKR Family.

Sets the size (in bits) of the value returned by `analogRead()`. It defaults to 10 bits (returns values between 0-1023) for backward compatibility with AVR based boards.

The **Due, Zero and MKR Family** boards have 12-bit ADC capabilities that can be accessed by changing the resolution to 12. This will return values from `analogRead()` between 0 and 4095.

Syntax

```
analogReadResolution(bits)
```

Parameters

`bits`: determines the resolution (in bits) of the value returned by the `analogRead()` function. You can set this between 1 and 32. You can set resolutions higher than 12 but values returned by `analogRead()` will suffer approximation. See the note below for details.

Returns

Nothing

Example Code

The code shows how to use ADC with different resolutions.

```
void setup() {
  // open a serial connection
  Serial.begin(9600);
}

void loop() {
  // read the input on A0 at default resolution (10 bits)
  // and send it out the serial connection
  analogReadResolution(10);
  Serial.print("ADC 10-bit (default) : ");
  Serial.print(analogRead(A0));

  // change the resolution to 12 bits and read A0
  analogReadResolution(12);
  Serial.print(", 12-bit : ");
  Serial.print(analogRead(A0));

  // change the resolution to 16 bits and read A0
  analogReadResolution(16);
  Serial.print(", 16-bit : ");
  Serial.print(analogRead(A0));

  // change the resolution to 8 bits and read A0
  analogReadResolution(8);
  Serial.print(", 8-bit : ");
  Serial.println(analogRead(A0));

  // a little delay to not hog Serial Monitor
  delay(100);
}
```

Notes and Warnings

If you set the `analogReadResolution()` value to a value higher than your board's capabilities, the Arduino will only report back at its highest resolution, padding the extra bits with zeros.

For example: using the Due with `analogReadResolution(16)` will give you an approximated 16-bit number with the first 12 bits containing the real ADC reading and the last 4 bits **padded with zeros**.

If you set the `analogReadResolution()` value to a value lower than your board's capabilities, the extra least significant bits read from the ADC will be **discarded**.

Using a 16 bit resolution (or any resolution **higher** than actual hardware capabilities) allows you to write sketches that automatically handle devices with a higher resolution ADC when these become available on future boards without changing a line of code.

analogWriteResolution() [Zero, Due & MKR Family]

Description

`analogWriteResolution()` is an extension of the Analog API for the Arduino Due.

`analogWriteResolution()` sets the resolution of the `analogWrite()` function. It defaults to 8 bits (values between 0-255) for backward compatibility with AVR based boards.

The **Due** has the following hardware capabilities:

- 12 pins which default to 8-bit PWM, like the AVR-based boards. These can be changed to 12-bit resolution.
- 2 pins with 12-bit DAC (Digital-to-Analog Converter)

By setting the write resolution to 12, you can use `analogWrite()` with values between 0 and 4095 to exploit the full DAC resolution or to set the PWM signal without rolling over.

The **Zero** has the following hardware capabilities:

- 10 pins which default to 8-bit PWM, like the AVR-based boards. These can be changed to 12-bit resolution.
- 1 pin with 10-bit DAC (Digital-to-Analog Converter).

By setting the write resolution to 10, you can use `analogWrite()` with values between 0 and 1023 to exploit the full DAC resolution

The **MKR Family** of boards has the following hardware capabilities:

- 4 pins which default to 8-bit PWM, like the AVR-based boards. These can be changed from 8 (default) to 12-bit resolution.
- 1 pin with 10-bit DAC (Digital-to-Analog Converter)

By setting the write resolution to 12 bits, you can use `analogWrite()` with values between 0 and 4095 for PWM signals; set 10 bit on the DAC pin to exploit the full DAC resolution of 1024 values.

Syntax

```
analogWriteResolution(bits)
```

Parameters

`bits`: determines the resolution (in bits) of the values used in the `analogWrite()` function. The value can range from 1 to 32. If you choose a resolution higher or lower than your board's hardware capabilities, the value used in `analogWrite()` will be either truncated if it's too high or padded with zeros if it's too low. See the note below for details.

Returns

Nothing

Example Code

Explain Code

```
void setup(){
  // open a serial connection
  Serial.begin(9600);
  // make our digital pin an output
  pinMode(11, OUTPUT);
  pinMode(12, OUTPUT);
  pinMode(13, OUTPUT);
}

void loop(){
  // read the input on A0 and map it to a PWM pin
  // with an attached LED
  int sensorVal = analogRead(A0);
  Serial.print("Analog Read) : ");
  Serial.print(sensorVal);

  // the default PWM resolution
  analogWriteResolution(8);
  analogWrite(11, map(sensorVal, 0, 1023, 0 ,255));
  Serial.print(" , 8-bit PWM value : ");
  Serial.print(map(sensorVal, 0, 1023, 0 ,255));

  // change the PWM resolution to 12 bits
  // the full 12 bit resolution is only supported
  // on the 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));

  // change the PWM resolution to 4 bits
  analogWriteResolution(4);
  analogWrite(13, map(sensorVal, 0, 1023, 0, 15));
  Serial.print(", 4-bit PWM value : ");
  Serial.println(map(sensorVal, 0, 1023, 0, 15));

  delay(5);
}
```

Notes and Warnings

If you set the `analogWriteResolution()` value to a value higher than your board's capabilities, the Arduino will discard the extra bits. For example: using the Due with `analogWriteResolution(16)` on a 12-bit DAC pin, only the first 12 bits of the values passed to `analogWrite()` will be used and the last 4 bits will be discarded.

If you set the `analogWriteResolution()` value to a value lower than your board's capabilities, the missing bits will be **padded with zeros** to fill the hardware required size. For example: using the Due with analogWriteResolution(8) on a 12-bit DAC pin, the Arduino will add 4 zero bits to the 8-bit value used in `analogWrite()` to obtain the 12 bits required.

Advanced I/O

tone() [Advanced I/O]

Description

Generates a square wave of the specified frequency (and 50% duty cycle) on a pin. A duration can be specified, otherwise the wave continues until a call to noTone(). The pin can be connected to a piezo buzzer or other speaker to play tones.

Only one tone can be generated at a time. If a tone is already playing on a different pin, the call to tone() will have no effect. If the tone is playing on the same pin, the call will set its frequency.

Use of the `tone()` function will interfere with PWM output on pins 3 and 11 (on boards other than the Mega).

It is not possible to generate tones lower than 31Hz. For technical details, see Brett Hagman's notes.

Syntax

```
tone(pin, frequency)
tone(pin, frequency, duration)
```

Parameters

`pin`: the pin on which to generate the tone
`frequency`: the frequency of the tone in hertz - `unsigned int`
`duration`: the duration of the tone in milliseconds (optional) - `unsigned long`

Returns

Nothing

Notes and Warnings

If you want to play different pitches on multiple pins, you need to call `noTone()` on one pin before calling `tone()` on the next pin.

noTone() [Advanced I/O]

Description

Stops the generation of a square wave triggered by `tone()`. Has no effect if no tone is being generated.

Syntax

```
noTone(pin)
```

Parameters

`pin`: the pin on which to stop generating the tone

Returns

Nothing

Notes and Warnings

If you want to play different pitches on multiple pins, you need to call noTone() on one pin before calling `tone()` on the next pin.

shiftOut() [Advanced I/O]

Description

Shifts out a byte of data one bit at a time. Starts from either the most (i.e. the leftmost) or least (rightmost) significant bit. Each bit is written in turn to a data pin, after which a clock pin is pulsed (taken high, then low) to indicate that the bit is available.

Note- if you're interfacing with a device that's clocked by rising edges, you'll need to make sure that the clock pin is low before the call to `shiftOut()`, e.g. with a call to `digitalWrite(clockPin, LOW)`.

This is a software implementation; see also the SPI library, which provides a hardware implementation that is faster but works only on specific pins.

Syntax

```
shiftOut(dataPin, clockPin, bitOrder, value)
```

Parameters

`dataPin`: the pin on which to output each bit (int)

`clockPin`: the pin to toggle once the dataPin has been set to the correct value (int)

`bitOrder`: which order to shift out the bits; either MSBFIRST or LSBFIRST. (Most Significant Bit First, or, Least Significant Bit First)

`value`: the data to shift out. (byte)

Returns

Nothing

Example Code

For accompanying circuit, see the tutorial on controlling a 74HC595 shift register.

```
//**************************************************************//
//  Name    : shiftOutCode, Hello World                         //
//  Author  : Carlyn Maw,Tom Igoe                               //
//  Date    : 25 Oct, 2006                                      //
//  Version : 1.0                                               //
//  Notes   : Code for using a 74HC595 Shift Register           //
//          : to count from 0 to 255                            //
//****************************************************************

//Pin connected to ST_CP of 74HC595
int latchPin = 8;
//Pin connected to SH_CP of 74HC595
int clockPin = 12;
////Pin connected to DS of 74HC595
int dataPin = 11;

void setup() {
  //set pins to output because they are addressed in the main loop
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}

void loop() {
  //count up routine
  for (int j = 0; j < 256; j++) {
    //ground latchPin and hold low for as long as you are transmitting
    digitalWrite(latchPin, LOW);
    shiftOut(dataPin, clockPin, LSBFIRST, j);
    //return the latch pin high to signal chip that it
    //no longer needs to listen for information
    digitalWrite(latchPin, HIGH);
    delay(1000);
  }
}
```

Notes and Warnings

The dataPin and clockPin must already be configured as outputs by a call to pinMode().

shiftOut is currently written to output 1 byte (8 bits) so it requires a two step operation to output values larger than 255.

```
// Do this for MSBFIRST serial
int data = 500;
// shift out highbyte
shiftOut(dataPin, clock, MSBFIRST, (data >> 8));
// shift out lowbyte
shiftOut(dataPin, clock, MSBFIRST, data);

// Or do this for LSBFIRST serial
data = 500;
// shift out lowbyte
shiftOut(dataPin, clock, LSBFIRST, data);
// shift out highbyte
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

shiftIn() [Advanced I/O]

Description

Shifts in a byte of data one bit at a time. Starts from either the most (i.e. the leftmost) or least (rightmost) significant bit. For each bit, the clock pin is pulled high, the next bit is read from the data line, and then the clock pin is taken low.

If you're interfacing with a device that's clocked by rising edges, you'll need to make sure that the clock pin is low before the first call to `shiftIn()`, e.g. with a call to `digitalWrite(clockPin, LOW)`.

Note: this is a software implementation; Arduino also provides an SPI library that uses the hardware implementation, which is faster but only works on specific pins.

Syntax

```
byte incoming = shiftIn(dataPin, clockPin, bitOrder)
```

Parameters

`dataPin`: the pin on which to input each bit (int)

`clockPin`: the pin to toggle to signal a read from **dataPin**

`bitOrder`: which order to shift in the bits; either **MSBFIRST** or **LSBFIRST**. (Most Significant Bit First, or, Least Significant Bit First)

Returns

the value read (byte)

pulseIn() [Advanced I/O]

Description

Reads a pulse (either HIGH or LOW) on a pin. For example, if **value** is **HIGH**, `pulseIn()` waits for the pin to go **HIGH**, starts timing, then waits for the pin to go **LOW** and stops timing. Returns the length of the pulse in microseconds. Gives up and returns 0 if no pulse starts within a specified time out.

The timing of this function has been determined empirically and will probably show errors in longer pulses. Works on pulses from 10 microseconds to 3 minutes in length.

Syntax

```
pulseIn(pin, value)
```

`pulseIn(pin, value, timeout)`

Parameters

`pin`: the number of the pin on which you want to read the pulse. (int)

`value`: type of pulse to read: either HIGH or LOW. (int)

`timeout` (optional): the number of microseconds to wait for the pulse to start; default is one second (unsigned long)

Returns

the length of the pulse (in microseconds) or 0 if no pulse started before the timeout (unsigned long)

Example Code

The example calculated the time duration of a pulse on pin 7.

```
int pin = 7;
unsigned long duration;

void setup()
{
  pinMode(pin, INPUT);
}

void loop()
{
  duration = pulseIn(pin, HIGH);
}
```

pulseInLong() [Advanced I/O]

Description

Reads a pulse (either HIGH or LOW) on a pin. For example, if value is HIGH, `pulseInLong()` waits for the pin to go `HIGH`, starts timing, then waits for the pin to go `LOW` and stops timing. Returns the length of the pulse in microseconds or 0 if no complete pulse was received within the timeout.

The timing of this function has been determined empirically and will probably show errors in shorter pulses. Works on pulses from 10 microseconds to 3 minutes in length. Please also note that if the pin is already high when the function is called, it will wait for the pin to go LOW and then HIGH before it starts counting. This routine can be used only if interrupts are activated. Furthermore the highest resolution is obtained with large intervals.

Syntax

`pulseInLong(pin, value)`

`pulseInLong(pin, value, timeout)`

Parameters

`pin`: the number of the pin on which you want to read the pulse. (int)

`value`: type of pulse to read: either HIGH or LOW. (int)

`timeout` (optional): the number of microseconds to wait for the pulse to start; default is one second (unsigned long)

Returns

the length of the pulse (in microseconds) or 0 if no pulse started before the timeout (unsigned long)

Example Code

The example calculated the time duration of a pulse on pin 7.

```
int pin = 7;
unsigned long duration;

void setup() {
  pinMode(pin, INPUT);
}

void loop() {
  duration = pulseInLong(pin, HIGH);
}
```

Notes and Warnings

This function relies on micros() so cannot be used in noInterrupts() context.

Time

millis() [Time]

Description

Returns the number of milliseconds since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 50 days.

Syntax

`time = millis()`

Parameters

Nothing

Returns

Number of milliseconds since the program started (unsigned long)

Example Code

The code reads the milllisecond since the Arduino board began.

```
unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = millis();

  Serial.println(time);    //prints time since program started
  delay(1000);             // wait a second so as not to send massive amounts of
data
}
```

Notes and Warnings

Please note that the return value for millis() is an unsigned long, logic errors may occur if a programmer tries to do arithmetic with smaller data types such as int's. Even signed long may encounter errors as its maximum value is half that of its unsigned counterpart.

micros() [Time]

Description

Returns the number of microseconds since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 70 minutes. On 16 MHz Arduino boards (e.g. Duemilanove and Nano), this function has a resolution of four microseconds (i.e. the value returned is always a multiple of four). On 8 MHz Arduino boards (e.g. the LilyPad), this function has a resolution of eight microseconds.

Syntax

```
time = micros()
```

Parameters

Nothing

Returns

Returns the number of microseconds since the Arduino board began running the current program.(unsigned long)

Example Code

The code returns the number of microseconds since the Arduino board began.

```
unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = micros();

  Serial.println(time);  //prints time since program started
  delay(1000);           // wait a second so as not to send massive amounts of
data
}
```

Notes and Warnings

There are 1,000 microseconds in a millisecond and 1,000,000 microseconds in a second.

delay() [Time]

Description

Pauses the program for the amount of time (in milliseconds) specified as parameter. (There are 1000 milliseconds in a second.)

Syntax

`delay(ms)`

Parameters

`ms: the number of milliseconds to pause (unsigned long)`

Returns

`Nothing`

Example Code

The code pauses the program for one second before toggling the output pin.

```
int ledPin = 13;                 // LED connected to digital pin 13

void setup()
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH);   // sets the LED on
  delay(1000);                  // waits for a second
  digitalWrite(ledPin, LOW);    // sets the LED off
  delay(1000);                  // waits for a second
}
```

Notes and Warnings

While it is easy to create a blinking LED with the `delay()` function, and many sketches use short delays for such tasks as switch debouncing, the use of `delay()` in a sketch has significant drawbacks. No other reading of sensors, mathematical calculations, or pin manipulation can go on during the delay function, so in effect, it brings most other activity to a halt. For alternative approaches to controlling timing see the millis() function and the sketch sited below. More knowledgeable programmers usually avoid the use of `delay()` for timing of events longer than 10's of milliseconds unless the Arduino sketch is very simple.

Certain things do go on while the delay() function is controlling the Atmega chip however, because the delay function does not disable interrupts. Serial communication that appears at the RX pin is recorded, PWM (analogWrite) values and pin states are maintained, and interrupts will work as they should.

delayMicroseconds() [Time]

Description

Pauses the program for the amount of time (in microseconds) specified as parameter. There are a thousand microseconds in a millisecond, and a million microseconds in a second.

Currently, the largest value that will produce an accurate delay is 16383. This could change in future Arduino releases. For delays longer than a few thousand microseconds, you should use `delay()` instead.

Syntax

`delayMicroseconds(us)`

Parameters

`us`: the number of microseconds to pause (`unsigned int`)

Returns

Nothing

Example Code

The code configures pin number 8 to work as an output pin. It sends a train of pulses of approximately 100 microseconds period. The approximation is due to execution of the other instructions in the code.

```
int outPin = 8;                 // digital pin 8

void setup()
{
  pinMode(outPin, OUTPUT);      // sets the digital pin as output
}

void loop()
```

```
{
  digitalWrite(outPin, HIGH);   // sets the pin on
  delayMicroseconds(50);        // pauses for 50 microseconds
  digitalWrite(outPin, LOW);    // sets the pin off
  delayMicroseconds(50);        // pauses for 50 microseconds
}
```

Notes and Warnings

This function works very accurately in the range 3 microseconds and up. We cannot assure that delayMicroseconds will perform precisely for smaller delay-times.

As of Arduino 0018, delayMicroseconds() no longer disables interrupts.

Math

min() [Math]

Description

Calculates the minimum of two numbers.

Syntax

`min(x, y)`

Parameters

`x`: the first number, any data type

`y`: the second number, any data type

Returns

The smaller of the two numbers.

Example Code

The code ensures that it never gets above 100.

```
sensVal = min(sensVal, 100); // assigns sensVal to the smaller of sensVal or 100
                             // ensuring that it never gets above 100.
```

Notes and Warnings

Perhaps counter-intuitively, `max()` is often used to constrain the lower end of a variable's range, while `min()` is used to constrain the upper end of the range.

Because of the way the `min()` function is implemented, avoid using other functions inside the brackets, it may lead to incorrect results

```
min(a++, 100);   // avoid this - yields incorrect results

min(a, 100);
a++;    // use this instead - keep other math outside the function
```

max() [Math]

Description

Calculates the maximum of two numbers.

Syntax

```
max(x, y)
```

Parameters

`x`: the first number, any data type `y`: the second number, any data type

Returns

The larger of the two parameter values.

Example Code

The code ensures that sensVal is at least 20.

```
sensVal = max(sensVal, 20); // assigns sensVal to the larger of sensVal or 20
                           // (effectively ensuring that it is at least 20)
```

Notes and Warnings

Perhaps counter-intuitively, `max()` is often used to constrain the lower end of a variable's range, while `min()` is used to constrain the upper end of the range.

Because of the way the `max()` function is implemented, avoid using other functions inside the brackets, it may lead to incorrect results

```
max(a--, 0);   // avoid this - yields incorrect results

max(a, 0);            // use this instead -
a--;     // keep other math outside the function
```

abs() [Math]

Description

Calculates the absolute value of a number.

Syntax

```
abs(x)
```

Parameters

`x`: the number

Returns

`x`: if x is greater than or equal to 0.

`-x`: if x is less than 0.

Notes and Warnings

Because of the way the abs() function is implemented, avoid using other functions inside the brackets, it may lead to incorrect results.

```
abs(a++);   // avoid this - yields incorrect results

abs(a);         // use this instead -
a++;        // keep other math outside the function
```

constrain() [Math]

Description

Constrains a number to be within a range.

Syntax

```
constrain(x, a, b)
```

Parameters

`x`: the number to constrain, all data types `a`: the lower end of the range, all data types `b`: the upper end of the range, all data types

Returns

x: if x is between a and b
a: if x is less than a
b: if x is greater than b

Example Code

The code limits the sensor values to between 10 to 150.

```
sensVal = constrain(sensVal, 10, 150);    // limits range of sensor values to
between 10 and 150
```

map() [Math]

Description

Re-maps a number from one range to another. That is, a value of **fromLow** would get mapped to **toLow**, a value of **fromHigh** to **toHigh**, values in-between to values in-between, etc.

Does not constrain values to within the range, because out-of-range values are sometimes intended and useful. The `constrain()` function may be used either before or after this function, if limits to the ranges are desired.

Note that the "lower bounds" of either range may be larger or smaller than the "upper bounds" so the `map()` function may be used to reverse a range of numbers, for example

```
y = map(x, 1, 50, 50, 1);
```

The function also handles negative numbers well, so that this example

```
y = map(x, 1, 50, 50, -100);
```

is also valid and works well.

The `map()` function uses integer math so will not generate fractions, when the math might indicate that it should do so. Fractional remainders are truncated, and are not rounded or averaged.

Syntax

```
map(value, fromLow, fromHigh, toLow, toHigh)
```

Parameters

`value`: the number to map
`fromLow`: the lower bound of the value’s current range
`fromHigh`: the upper bound of the value’s current range
`toLow`: the lower bound of the value’s target range
`toHigh`: the upper bound of the value’s target range

Returns

The mapped value.

Example Code

```
/* Map an analog value to 8 bits (0 to 255) */
void setup() {}

void loop()
{
  int val = analogRead(0);
  val = map(val, 0, 1023, 0, 255);
  analogWrite(9, val);
}
```

Appendix

For the mathematically inclined, here's the whole function

```
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;
}
```

pow() [Math]

Description

Calculates the value of a number raised to a power. `Pow()` can be used to raise a number to a fractional power. This is useful for generating exponential mapping of values or curves.

Syntax

`pow(base, exponent)`

Parameters

`base`: the number (`float`)

`exponent`: the power to which the base is raised (`float`)

Returns

The result of the exponentiation. (`double`)

Example Code

See the (fscale) function in the code library.

sqrt() [Math]

Calculates the square root of a number.

Description

Syntax

`sqrt(x)`

Parameters

`x`: the number, any data type

Returns

The number’s square root. (double)

sq() [Math]

Description

Calculates the square of a number: the number multiplied by itself.

Syntax

`sq(x)`

Parameters

`x`: the number, any data type

Returns

The square of the number. (double)

Trigonometry

sin() [Trigonometry]

Description

Calculates the sine of an angle (in radians). The result will be between -1 and 1.

Syntax

`sin(rad)`

Parameters

`rad`: The angle in Radians (`float`).

Returns

The sine of the angle (`double`).

cos() [Trigonometry]

Description

Calculates the cosine of an angle (in radians). The result will be between -1 and 1.

Syntax

```
cos(rad)
```

Parameters

`rad`: The angle in Radians (float).

Returns

The cos of the angle (`double`).

tan() [Trigonometry]

Description

Calculates the tangent of an angle (in radians). The result will be between negative infinity and infinity.

Syntax

```
tan(rad)
```

Parameters

`rad`: The angle in Radians (`float`).

Returns

The tangent of the angle (`double`).

Characters

isAlphaNumeric() [Characters]

Description

Analyse if a char is alphanumeric (that is a letter or a numbers). Returns true if thisChar contains either a number or a letter.

Syntax

`isAlphaNumeric(thisChar)`

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is alphanumeric.

Example Code

```
if (isAlphaNumeric(this))       // tests if this isa letter or a number
{
        Serial.println("The character is alphanumeric");
}
else
{
        Serial.println("The character is not alphanumeric");
}
```

isAlpha()   [Characters]

Description

Analyse if a char is alpha (that is a letter). Returns true if thisChar contains a letter.

Syntax

`isAlpha(thisChar)`

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is alpha.

Example Code

```
if (isAlpha(this))       // tests if this is a letter
{
        Serial.println("The character is a letter");
}
else
{
        Serial.println("The character is not a letter");
}
```

isAscii() [Characters]

Description

Analyse if a char is Ascii. Returns true if thisChar contains an Ascii character.

Syntax

`` `isAscii(thisChar)` ``

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is Ascii.

Example Code

```
if (isAscii(this))      // tests if this is an Ascii character
{
        Serial.println("The character is Ascii");
}
else
{
        Serial.println("The character is not Ascii");
}
```

isWhitespace() [Characters]

Description

Analyse if a char is a white space, that is space, formfeed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v')). Returns true if thisChar contains a white space.

Syntax

`` `isWhitespace(thisChar)` ``

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is a white space.

Example Code

```
if (isWhitespace(this))      // tests if this is a white space
{
        Serial.println("The character is a white space");
}
else
{
        Serial.println("The character is not a white space");
}
```

isControl() [Characters]

Description

Analyse if a char is a control character. Returns true if thisChar is a control character.

Syntax

`` `isControl(thisChar)` ``

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is a control character.

Example Code

```
if (isControl(this))      // tests if this is a control character
{
        Serial.println("The character is a control character");
}
else
{
        Serial.println("The character is not a control character");
}
```

isDigit() [Characters]

Description

Analyse if a char is a digit (that is a number). Returns true if thisChar is a number.

Syntax

`isDigit(thisChar)`

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is a number.

Example Code

```
if (isDigit(this))      // tests if this is a digit
{
        Serial.println("The character is a number");
}
else
{
        Serial.println("The character is not a number");
}
```

isGraph() [Characters]

Description

Analyse if a char is printable with some content (space is printable but has no content). Returns true if thisChar is printable.

Syntax

```
`isGraph(thisChar)`
```

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is printable.

Example Code

```
if (isGraph(this))      // tests if this is a printable character but not a
blank space.
{
        Serial.println("The character is printable");
}
else
{
        Serial.println("The character is not printable");
}
```

isLowerCase() [Characters]

Description

Analyse if a char is lower case (that is a letter in lower case). Returns true if thisChar contains a letter in lower case.

Syntax

`` `isLowerCase(thisChar)` ``

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is lower case.

Example Code

```
if (isLowerCase(this))      // tests if this is a lower case letter
{
        Serial.println("The character is lower case");
}
else
{
        Serial.println("The character is not lower case");
}
```

isPrintable() [Characters]

Description

Analyse if a char is printable (that is any character that produces an output, even a blank space). Returns true if thisChar is printable.

Syntax

`` `isAlpha(thisChar)` ``

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is printable.

Example Code

```
if (isPrintable(this))      // tests if this is printable char
{
        Serial.println("The character is printable");
}
else
{
        Serial.println("The character is not printable");
}
```

isPunct()   [Characters]

Description

Analyse if a char is punctuation (that is a comma, a semicolon, an exlamation mark and so on). Returns true if thisChar is punctuation.

Syntax

`` `isPunct(thisChar)` ``

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is a punctuation.

Example Code

```
if (isPunct(this))      // tests if this is a punctuation character
{
        Serial.println("The character is a punctuation");
}
else
{
        Serial.println("The character is not a punctuation");
}
```

isSpace()   [Characters]

Description

Analyse if a char is the space character. Returns true if thisChar contains the space character.

Syntax

`` `isSpace(thisChar)` ``

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is a space.

Example Code

```
if (isSpace(this))      // tests if this is the space character
{
        Serial.println("The character is a space");
}
else
{
        Serial.println("The character is not a space");
}
```

isUpperCase() [Characters]

Description

Analyse if a char is upper case (that is a letter in upper case). Returns true if thisChar is upper case.

Syntax

``isUpperCase(thisChar)``

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is upper case.

Example Code

```
if (isUpperCase(this))      // tests if this is an upeer case letter
{
        Serial.println("The character is upper case");
}
else
{
        Serial.println("The character is not upper case");
}
```

isHexadecimalDigit() [Characters]

Description

Analyse if a char is an hexadecimal digit (A-F, 0-9). Returns true if thisChar contains an hexadecimal digit.

Syntax

`` `isHexadecimalDigit(thisChar)` ``

Parameters

`thisChar`: variable. **Allowed data types:** char

Returns

`true`: if thisChar is an hexadecimal digit.

Example Code

```
if (isHexadecimalDigit(this))      // tests if this is an hexadecimal digit
{
        Serial.println("The character is an hexadecimal digit");
}
else
{
        Serial.println("The character is not an hexadecimal digit");

}
```

Random Numbers

randomSeed() [Random Numbers]

Description

`randomSeed()` initializes the pseudo-random number generator, causing it to start at an arbitrary point in its random sequence. This sequence, while very long, and random, is always the same.

If it is important for a sequence of values generated by `random()` to differ, on subsequent executions of a sketch, use randomSeed() to initialize the random number generator with a fairly random input, such as `analogRead()` on an unconnected pin.

Conversely, it can occasionally be useful to use pseudo-random sequences that repeat exactly. This can be accomplished by calling `randomSeed()` with a fixed number, before starting the random sequence.

Parameters

`seed` - number to initialize the pseudo-random sequence (unsigned long).

Returns

Nothing

Example Code

The code explanation required.

```
long randNumber;

void setup(){
  Serial.begin(9600);
  randomSeed(analogRead(0));
}

void loop(){
  randNumber = random(300);
  Serial.println(randNumber);

  delay(50);
}
```

random() [Random Numbers]

Description

The random function generates pseudo-random numbers.

Syntax

```
random(max)
random(min, max)
```

Parameters

`min` - lower bound of the random value, inclusive (optional)

`max` - upper bound of the random value, exclusive

Returns

A random number between min and max-1 (`long`) .

Example Code

The code generates random numbers and displays them.

```
long randNumber;

void setup(){
  Serial.begin(9600);
  // if analog input pin 0 is unconnected, random analog
  // noise will cause the call to randomSeed() to generate
  // different seed numbers each time the sketch runs.
  // randomSeed() will then shuffle the random function.
  randomSeed(analogRead(0));
}

void loop() {
  // print a random number from 0 to 299
  randNumber = random(300);
  Serial.println(randNumber);

  // print a random number from 10 to 19
  randNumber = random(10, 20);
  Serial.println(randNumber);

  delay(50);
}
```

Notes and Warnings

If it is important for a sequence of values generated by `random()` to differ, on subsequent executions of a sketch, use `randomSeed()` to initialize the random number generator with a fairly random input, such as `analogRead()` on an unconnected pin.

Conversely, it can occasionally be useful to use pseudo-random sequences that repeat exactly. This can be accomplished by calling `randomSeed()` with a fixed number, before starting the random sequence.

The `max` parameter should be chosen according to the data type of the variable in which the value is stored. In any case, the absolute maximum is bound to the `long` nature of the value generated (32 bit - 2,147,483,647). Setting `max` to a higher value won’t generate an error during compilation, but during sketch execution the numbers generated will not be as expected.

Bits and Bytes

lowByte() [Bits and Bytes]

Description

Extracts the low-order (rightmost) byte of a variable (e.g. a word).

Syntax

`lowByte(x)`

Parameters

`x`: a value of any type

Returns

byte

highByte() [Bits and Bytes]

Description

Extracts the high-order (leftmost) byte of a word (or the second lowest byte of a larger data type).

Syntax

```
highByte(x)
```

Parameters

`x`: a value of any type

Returns

byte

bitRead() [Bits and Bytes]

Description

Reads a bit of a number.

Syntax

```
bitRead(x, n)
```

Parameters

`x`: the number from which to read

`n`: which bit to read, starting at 0 for the least-significant (rightmost) bit

Returns

the value of the bit (0 or 1).

bitWrite() [Bits and Bytes]

Description

Writes a bit of a numeric variable.

Syntax

```
bitWrite(x, n, b)
```

Parameters

`x`: the numeric variable to which to write

`n`: which bit of the number to write, starting at 0 for the least-significant (rightmost) bit

`b`: the value to write to the bit (0 or 1)

Returns

Nothing

bitSet() [Bits and Bytes]

Sets (writes a 1 to) a bit of a numeric variable.

Description

Syntax

```
bitSet(x, n)
```

Parameters

`x`: the numeric variable whose bit to set
`n`: which bit to set, starting at 0 for the least-significant (rightmost) bit

Returns

Nothing

bitClear() [Bits and Bytes]

Description

Clears (writes a 0 to) a bit of a numeric variable.

Syntax

```
bitClear(x, n)
```

Parameters

`x`: the numeric variable whose bit to clear

`n`: which bit to clear, starting at 0 for the least-significant (rightmost) bit

Returns

Nothing

bit() [Bits and Bytes]

Description

Computes the value of the specified bit (bit 0 is 1, bit 1 is 2, bit 2 is 4, etc.).

Syntax

```
bit(n)
```

Parameters

`n`: the bit whose value to compute

Returns

The value of the bit.

External Interrupts

attachInterrupt() [External Interrupts]

Description

Digital Pins With Interrupts

The first parameter to attachInterrupt is an interrupt number. Normally you should use digitalPinToInterrupt(pin) to translate the actual digital pin to the specific interrupt number. For example, if you connect to pin 3, use digitalPinToInterrupt(3) as the first parameter to attachInterrupt.

Board	Digital Pins Usable For Interrupts
Uno, Nano, Mini, other 328-based	2, 3
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21
Micro, Leonardo, other 32u4-based	0, 1, 2, 3, 7
Zero	all digital pins, except 4
MKR1000 Rev.1	0, 1, 4, 5, 6, 7, 8, 9, A1, A2

Board	Digital Pins Usable For Interrupts
Due	all digital pins
101	all digital pins (Only pins 2, 5, 7, 8, 10, 11, 12, 13 work with **CHANGE**)

Notes and Warnings

Note
Inside the attached function, `delay()` won't work and the value returned by `millis()` will not increment. Serial data received while in the function may be lost. You should declare as volatile any variables that you modify within the attached function. See the section on ISRs below for more information.

Using Interrupts

Interrupts are useful for making things happen automatically in microcontroller programs, and can help solve timing problems. Good tasks for using an interrupt may include reading a rotary encoder, or monitoring user input.

If you wanted to insure that a program always caught the pulses from a rotary encoder, so that it never misses a pulse, it would make it very tricky to write a program to do anything else, because the program would need to constantly poll the sensor lines for the encoder, in order to catch pulses when they occurred. Other sensors have a similar interface dynamic too, such as trying to read a sound sensor that is trying to catch a click, or an infrared slot sensor (photo-interrupter) trying to catch a coin drop. In all of these situations, using an interrupt can free the microcontroller to get some other work done while not missing the input.

About Interrupt Service Routines

ISRs are special kinds of functions that have some unique limitations most other functions do not have. An ISR cannot have any parameters, and they shouldn't return anything.

Generally, an ISR should be as short and fast as possible. If your sketch uses multiple ISRs, only one can run at a time, other interrupts will be executed after the current one finishes in an order that depends on the priority they have. millis() relies on interrupts to count, so it will never increment inside an ISR. Since delay() requires interrupts to work, it will not work if called inside an ISR. micros() works initially, but will start behaving erratically after 1-2 ms. delayMicroseconds() does not use any counter, so it will work as normal.

Typically global variables are used to pass data between an ISR and the main program. To make sure variables shared between an ISR and the main program are updated correctly, declare them as `volatile`.

For more information on interrupts, see Nick Gammon's notes.

Syntax

`attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);` (recommended)
`attachInterrupt(interrupt, ISR, mode);` (not recommended)
`attachInterrupt(pin, ISR, mode);` (not recommended Arduino Due, Zero, MKR1000, 101 only)

Parameters

`interrupt`: the number of the interrupt (`int`)
`pin`: the pin number *(Arduino Due, Zero, MKR1000 only)*
`ISR`: the ISR to call when the interrupt occurs; this function must take no parameters and return nothing. This function is sometimes referred to as an interrupt service routine.
`mode`: defines when the interrupt should be triggered. Four constants are predefined as valid values:

- **LOW** to trigger the interrupt whenever the pin is low,
- **CHANGE** to trigger the interrupt whenever the pin changes value
- **RISING** to trigger when the pin goes from low to high,
- **FALLING** for when the pin goes from high to low.
  The Due, Zero and MKR1000 boards allows also:
- **HIGH** to trigger the interrupt whenever the pin is high.

Returns

Nothing

Example Code

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

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

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

Interrupt Numbers

Normally you should use digitalPinToInterrupt(pin), rather than place an interrupt number directly into your sketch. The specific pins with interrupts, and their mapping to interrupt number varies on each type of board. Direct use of interrupt numbers may seem simple, but it can cause compatibility trouble when your sketch is run on a different board.

However, older sketches often have direct interrupt numbers. Often number 0 (for digital pin 2) or number 1 (for digital pin 3) were used. The table below shows the available interrupt pins on various boards.

Note that in the table below, the interrupt numbers refer to the number to be passed to attachInterrupt(). For historical reasons, this numbering does not always correspond directly to the interrupt numbering on the atmega chip (e.g. int.0 corresponds to INT4 on the Atmega2560 chip).

Board	int.0	int.1	int.2	int.3	int.4	int.5

Board	int.0	int.1	int.2	int.3	int.4	int.5
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
32u4 based (e.g Leonardo, Micro)	3	2	0	1	7	

For Due, Zero, MKR1000 and 101 boards the **interrupt number = pin number**.

detachInterrupt() [External Interrupts]

Description

Turns off the given interrupt.

Syntax

```
detachInterrupt()
detachInterrupt(pin)
```
(Arduino Due only)

Parameters

`interrupt`: the number of the interrupt to disable (see attachInterrupt() for more details).

`pin`: the pin number of the interrupt to disable (Arduino Due only)

Returns

Nothing

Interrupts

interrupts() [Interrupts]

Description

Re-enables interrupts (after they've been disabled by nointerrupts(). Interrupts allow certain important tasks to happen in the background and are enabled by default. Some functions will not work while interrupts are disabled, and incoming communication may be ignored. Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.

Syntax

```
interrupts()
```

Parameters

Nothing

Returns

Nothing

Example Code

The code enables Interrupts.

```
void setup() {}

void loop()
{
  noInterrupts();
  // critical, time-sensitive code here
  interrupts();
  // other code here
}
```

noInterrupts() [Interrupts]

Description

Disables interrupts (you can re-enable them with `interrupts()`). Interrupts allow certain important tasks to happen in the background and are enabled by default. Some functions will not work while interrupts are disabled, and incoming communication may be ignored. Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.

Syntax

`noInterrupts()`

Parameters

Nothing

Returns

Nothing

Example Code

The code shows how to enable interrupts.

```
void setup() {}

void loop()
```

```
{
noInterrupts();
// critical, time-sensitive code here
interrupts();
// other code here
}
```

Communication

Serial [Communication]

Description

Used for communication between the Arduino board and a computer or other devices. All Arduino boards have at least one serial port (also known as a UART or USART): Serial. It communicates on digital pins 0 (RX) and 1 (TX) as well as with the computer via USB. Thus, if you use these functions, you cannot also use pins 0 and 1 for digital input or output.
You can use the Arduino environment's built-in serial monitor to communicate with an Arduino board. Click the serial monitor button in the toolbar and select the same baud rate used in the call to `begin()`.

Serial communication on pins TX/RX uses TTL logic levels (5V or 3.3V depending on the board). Don't connect these pins directly to an RS232 serial port; they operate at +/- 12V and can damage your Arduino board.

The **Arduino Mega** has three additional serial ports: `Serial1` on pins 19 (RX) and 18 (TX), `Serial2` on pins 17 (RX) and 16 (TX), `Serial3` on pins 15 (RX) and 14 (TX). To use these pins to communicate with your personal computer, you will need an additional USB-to-serial adaptor, as they are not connected to the Mega's USB-to-serial adaptor. To use them to communicate with an external TTL serial device, connect the TX pin to your device's RX pin, the RX to your device's TX pin, and the ground of your Mega to your device's ground.

The **Arduino DUE** has three additional 3.3V TTL serial ports: `Serial1` on pins 19 (RX) and 18 (TX); `Serial2` on pins 17 (RX) and 16 (TX), `Serial3` on pins 15 (RX) and 14 (TX). Pins 0 and 1 are also connected to the corresponding pins of the ATmega16U2 USB-to-TTL Serial chip, which is connected to the USB debug port. Additionally, there is a native USB-serial port on the SAM3X chip, SerialUSB'.

The **Arduino Leonardo** board uses `Serial1` to communicate via TTL (5V) serial on pins 0 (RX) and 1 (TX). `Serial` is reserved for USB CDC communication. For more information, refer to the Leonardo getting started page and hardware page.

Functions

If (Serial)
available()
availableForWrite()
begin()
end()
find()
findUntil()
flush()
parseFloat()
parseInt()
peek()
print()
println()
read()
readBytes()
readBytesUntil()
setTimeout()
write()
serialEvent()

if(Serial)

Description

Indicates if the specified Serial port is ready.

On the Leonardo, `if (Serial)` indicates whether or not the USB CDC serial connection is open. For all other instances, including `if (Serial1)` on the Leonardo, this will always return true.

This was introduced in Arduino IDE 1.0.1.

Syntax

All boards:

```
if (Serial)
```

Arduino Leonardo specific:

```
if (Serial1)
```

Arduino Mega specific:

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

Parameters

Nothing

Returns

`boolean` : returns true if the specified serial port is available. This will only return false if querying the Leonardo’s USB CDC serial connection before it is ready.

Example Code

```
void setup() {
 //Initialize serial and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB
  }
}

void loop() {
 //proceed normally
}
```

Serial.available()

Description

Get the number of bytes (characters) available for reading from the serial port. This is data that's already arrived and stored in the serial receive buffer (which holds 64 bytes). `available()` inherits from the Stream utility class.

Syntax

```
Serial.available()
```

Arduino Mega only:

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

Parameters

None

Returns

The number of bytes available to read .

Example Code

The following code returns a character received through the serial port.

```
int incomingByte = 0;     // for incoming serial data

void setup() {
   Serial.begin(9600);    // opens serial port, sets data rate to 9600 bps
}

void loop() {

   // reply only when you receive data:
   if (Serial.available() > 0) {
           // read the incoming byte:
           incomingByte = Serial.read();

           // say what you got:
           Serial.print("I received: ");
           Serial.println(incomingByte, DEC);
   }
}
```

Arduino Mega example: This code sends data received in one serial port of the Arduino Mega to another. This can be used, for example, to connect a serial device to the computer through the Arduino board.

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

void loop() {
  // read from port 0, send to port 1:
  if (Serial.available()) {
    int inByte = Serial.read();
    Serial1.print(inByte, DEC);

  }
  // read from port 1, send to port 0:
  if (Serial1.available()) {
    int inByte = Serial1.read();
    Serial.print(inByte, DEC);
  }
}
```

Serial.availableForWrite()

Description

Get the number of bytes (characters) available for writing in the serial buffer without blocking the write operation.

Syntax

```
Serial.availableForWrite()
```

Arduino Mega only:

```
Serial1.availableForWrite()
Serial2.availableForWrite()
Serial3.availableForWrite()
```

Parameters

Nothing

Returns

The number of bytes available to write.

Serial.begin()

Description

Sets the data rate in bits per second (baud) for serial data transmission. For communicating with the computer, use one of these rates: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200. You can, however, specify other rates - for example, to communicate over pins 0 and 1 with a component that requires a particular baud rate.

An optional second argument configures the data, parity, and stop bits. The default is 8 data bits, no parity, one stop bit.

Syntax

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

Arduino Mega only:

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

Parameters

`speed`: in bits per second (baud) - `long`

`config`: sets data, parity, and stop bits. Valid values are

`SERIAL_5N1`
`SERIAL_6N1`
`SERIAL_7N1`
`SERIAL_8N1` (the default)
`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`

Returns

Nothing

Example Code

```
void setup() {
    Serial.begin(9600); // opens serial port, sets data rate to 9600 bps
}

void loop() {}
```

Arduino Mega example:

```
// Arduino Mega using all four of its Serial ports
// (Serial, Serial1, Serial2, Serial3),
// with different baud rates:

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() {}
```

Thanks to Jeff Gray for the mega example

Serial.end()

Description

Disables serial communication, allowing the RX and TX pins to be used for general input and output. To re-enable serial communication, call Serial.begin().

Syntax

```
Serial.end()
```

Arduino Mega only:

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

Parameters

Nothing

Returns

Nothing

Serial.find()

Description

Serial.find() reads data from the serial buffer until the target string of given length is found. The function returns true if target string is found, false if it times out.

Serial.find() inherits from the stream utility class.

Syntax

`Serial.find(target)`

Parameters

`target` : the string to search for (char)

Returns

`Boolean`

Serial.findUntil()

Description

`Serial.findUntil()` reads data from the serial buffer until a target string of given length or terminator string is found.

The function returns true if the target string is found, false if it times out.

`Serial.findUntil()` inherits from the Stream utility class.

Syntax

`Serial.findUntil(target, terminal)`

Parameters

`target` : the string to search for (char) `terminal` : the terminal string in the search (char)

Returns

`Boolean`

Serial.flush()

Description

Waits for the transmission of outgoing serial data to complete. (Prior to Arduino 1.0, this instead removed any buffered incoming serial data.)

`flush()` inherits from the Stream utility class.

Syntax

`Serial.flush()`

Arduino Mega only:

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

Parameters

Nothing

Returns

Nothing

Serial.parseFloat()

Description

`Serial.parseFloat()` returns the first valid floating point number from the Serial buffer. Characters that are not digits (or the minus sign) are skipped. `parseFloat()` is terminated by the first character that is not a floating point number.

`Serial.parseFloat()` inherits from the Stream utility class.

Syntax

```
Serial.parseFloat()
```

Parameters

Nothing

Returns

`Float`

Serial.parseInt()

Description

Looks for the next valid integer in the incoming serial `stream.parseInt()` inherits from the Stream utility class.
In particular:

- Initial characters that are not digits or a minus sign, are skipped;
- Parsing stops when no characters have been read for a configurable time-out value, or a non-digit is read;
- If no valid digits were read when the time-out (see Serial.setTimeout()) occurs, 0 is returned;

Syntax

```
Serial.parseInt() Serial.parseInt(char skipChar)
```

Arduino Mega only:

```
Serial1.parseInt()
Serial2.parseInt()
Serial3.parseInt()
```

Parameters

`skipChar`: used to skip the indicated char in the search. Used for example to skip thousands divider.

Returns

`long` : the next valid integer

Serial.peek()

Description

Returns the next byte (character) of incoming serial data without removing it from the internal serial buffer. That is, successive calls to `peek()` will return the same character, as will the next call to `read()`. `peek()` inherits from the Stream utility class.

Syntax

```
Serial.peek()
```

Arduino Mega only:

```
Serial1.peek()
Serial2.peek()
Serial3.peek()
```

Parameters

Nothing

Returns

The first byte of incoming serial data available (or -1 if no data is available) - `int`

Serial.print()

Description

Prints data to the serial port as human-readable ASCII text. This command can take many forms. Numbers are printed using an ASCII character for each digit. Floats are similarly printed as ASCII digits, defaulting to two decimal places. Bytes are sent as a single character. Characters and strings are sent as is. For example-

- `Serial.print(78) gives "78"`
- `Serial.print(1.23456) gives "1.23"`
- `Serial.print('N') gives "N"`
- `Serial.print("Hello world.") gives "Hello world." `

An optional second parameter specifies the base (format) to use; permitted values are `BIN(binary, or base 2)`, `OCT(octal, or base 8)`, `DEC(decimal, or base 10)`, `HEX(hexadecimal, or base 16)`. For floating point numbers, this parameter specifies the number of decimal places to use. For example-

- `Serial.print(78, BIN) gives "1001110"`
- `Serial.print(78, OCT) gives "116"`
- `Serial.print(78, DEC) gives "78"`
- `Serial.print(78, HEX) gives "4E"`
- `Serial.println(1.23456, 0) gives "1"`
- `Serial.println(1.23456, 2) gives "1.23"`
- `Serial.println(1.23456, 4) gives "1.2346"`

You can pass flash-memory based strings to Serial.print() by wrapping them with F(). For example:

```
Serial.print(F("Hello World"))
```

To send a single byte, use Serial.write().

Syntax

```
Serial.print(val)
Serial.print(val, format)
```

Parameters

`val`: the value to print - any data type

Returns

`size_t`: `print()` returns the number of bytes written, though reading that number is optional.

Example Code

```
/*
Uses a FOR loop for data and prints a number in various formats.
*/
int x = 0;    // variable

void setup() {
  Serial.begin(9600);      // open the serial port at 9600 bps:
}

void loop() {
  // print labels
  Serial.print("NO FORMAT");       // prints a label
  Serial.print("\t");              // prints a tab

  Serial.print("DEC");
  Serial.print("\t");

  Serial.print("HEX");
  Serial.print("\t");
  Serial.print("OCT");
  Serial.print("\t");

  Serial.print("BIN");
  Serial.println("\t");           // carriage return after the last label

  for(x=0; x< 64; x++){     // only part of the ASCII chart, change to suit

    // print it out in many formats:
    Serial.print(x);       // print as an ASCII-encoded decimal - same as "DEC"
    Serial.print("\t\t");  // prints two tabs to accomodate the label lenght

    Serial.print(x, DEC);  // print as an ASCII-encoded decimal
    Serial.print("\t");    // prints a tab

    Serial.print(x, HEX);  // print as an ASCII-encoded hexadecimal
    Serial.print("\t");    // prints a tab

    Serial.print(x, OCT);  // print as an ASCII-encoded octal
    Serial.print("\t");    // prints a tab

    Serial.println(x, BIN);  // print as an ASCII-encoded binary
    //                             then adds the carriage return with "println"
    delay(200);            // delay 200 milliseconds
  }
  Serial.println("");      // prints another carriage return
}
```

Notes and Warnings

As of version 1.0, serial transmission is asynchronous; `Serial.print()` will return before any characters are transmitted.

Serial.println()

Description

Prints data to the serial port as human-readable ASCII text followed by a carriage return character (ASCII 13, or '\r') and a newline character (ASCII 10, or '\n'). This command takes the same forms as Serial.print().

Syntax

```
Serial.println(val)
Serial.println(val, format)
```

Parameters

`val`: the value to print - any data type

`format`: specifies the number base (for integral data types) or number of decimal places (for floating point types)

Returns

`size_t`: `println()` returns the number of bytes written, though reading that number is optional

Example Code

```
/*
 Analog input reads an analog input on analog in 0, prints the value out.
 created 24 March 2006
 by Tom Igoe
 */

int analogValue = 0;    // variable to hold the analog value

void setup() {
  // open the serial port at 9600 bps:
  Serial.begin(9600);
}

void loop() {
  // read the analog input on pin 0:
  analogValue = analogRead(0);

  // print it out in many formats:
  Serial.println(analogValue);       // print as an ASCII-encoded decimal
  Serial.println(analogValue, DEC);  // print as an ASCII-encoded decimal
  Serial.println(analogValue, HEX);  // print as an ASCII-encoded hexadecimal
  Serial.println(analogValue, OCT);  // print as an ASCII-encoded octal
  Serial.println(analogValue, BIN);  // print as an ASCII-encoded binary

  // delay 10 milliseconds before the next reading:
  delay(10);
```

Serial.read()

Description

Reads incoming serial data. read() inherits from the Stream utility class.

Syntax

```
Serial.read()
```

Arduino Mega only:

```
Serial1.read()
Serial2.read()
Serial3.read()
```

Parameters

Nothing

Returns

The first byte of incoming serial data available (or -1 if no data is available) - `int`.

Example Code

```
int incomingByte = 0;   // for incoming serial data

void setup() {
        Serial.begin(9600);     // opens serial port, sets data rate to 9600 bps
}

void loop() {

        // send data only when you receive data:
        if (Serial.available() > 0) {
                // read the incoming byte:
                incomingByte = Serial.read();

                // say what you got:
                Serial.print("I received: ");
                Serial.println(incomingByte, DEC);
        }
}
```

Serial.readBytes()

Description

`Serial.readBytes()` reads characters from the serial port into a buffer. The function terminates if the determined length has been read, or it times out (see Serial.setTimeout()).

`Serial.readBytes()` returns the number of characters placed in the buffer. A 0 means no valid data was found.

`Serial.readBytes()` inherits from the Stream utility class.

Syntax

`Serial.readBytes(buffer, length)`

Parameters

`buffer`: the buffer to store the bytes in (`char[]` or `byte[]`)

`length` : the number of bytes to read (`int`)

Returns

The number of bytes placed in the buffer (`size_t`)

Serial.readBytesUntil()

Description

Serial.readBytesUntil() reads characters from the serial buffer into an array. The function terminates if the terminator character is detected, the determined length has been read, or it times out (see Serial.setTimeout()). The function returns the characters up to the last character before the supplied terminator. The terminator itself is not returned in the buffer.

`Serial.readBytesUntil()` returns the number of characters read into the buffer. A 0 means no valid data was found.

`Serial.readBytesUntil()` inherits from the Stream utility class.

Syntax

```
Serial.readBytesUntil(character, buffer, length)
```

Parameters

`character` : the character to search for (`char`)
`buffer`: the buffer to store the bytes in (`char[]` or `byte[]`)
`length` : the number of bytes to read (`int`)

Returns

`size_t`

Serial.setTimeout()

Description

`Serial.setTimeout()` sets the maximum milliseconds to wait for serial data when using serial.readBytesUntil() or serial.readBytes(). It defaults to 1000 milliseconds.

`Serial.setTimeout()` inherits from the Stream utility class.

Syntax

```
Serial.setTimeout(time)
```

Parameters

`time` : timeout duration in milliseconds (`long`).

Returns

Nothing

Serial.write()

Description

Writes binary data to the serial port. This data is sent as a byte or series of bytes; to send the characters representing the digits of a number use the print() function instead.

Syntax

```
Serial.write(val)
Serial.write(str)
Serial.write(buf, len)
```

Arduino Mega also supports:

`Serial1`, `Serial2`, `Serial3` (in place of `Serial`)

Parameters

`val`: a value to send as a single byte

`str`: a string to send as a series of bytes

`buf`: an array to send as a series of bytes

Returns

`size_t`

`write()` will return the number of bytes written, though reading that number is optional

Example Code

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

void loop(){
  Serial.write(45); // send a byte with the value 45

   int bytesSent = Serial.write(“hello”); //send the string “hello” and return
the length of the string.
}
```

Serial.serialEvent()

Description

Called when data is available. Use `Serial.read()` to capture this data.

NB : Currently, `serialEvent()` is not compatible with the Esplora, Leonardo, or Micro

Syntax

```
void serialEvent(){
//statements
}
```

Arduino Mega only:

```
void serialEvent1(){
//statements
}

void serialEvent2(){
//statements
}

void serialEvent3(){
//statements
}
```

Parameters

`statements`: any valid statements

Returns

Nothing

stream [Communication]

Description

Stream is the base class for character and binary based streams. It is not called directly, but invoked whenever you use a function that relies on it.

Stream defines the reading functions in Arduino. When using any core functionality that uses a `read()` or similar method, you can safely assume it calls on the Stream class. For functions like `print()`, Stream inherits from the Print class.

Some of the libraries that rely on Stream include :

- Serial
- Wire
- Ethernet
- SD

Wire Library

This library allows you to communicate with I2C / TWI devices. On the Arduino boards with the R3 layout (1.0 pinout), the SDA (data line) and SCL (clock line) are on the pin headers close to the AREF pin. The Arduino Due has two I2C / TWI interfaces SDA1 and SCL1 are near to the AREF pin and the additional one is on pins 20 and 21.

As a reference the table below shows where TWI pins are located on various Arduino boards.

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

As of Arduino 1.0, the library inherits from the Stream functions, making it consistent with other read/write libraries. Because of this, send() and receive() have been replaced with read() and write().

Note

There are both 7- and 8-bit versions of I2C addresses. 7 bits identify the device, and the eighth bit determines if it's being written to or read from. The Wire library uses 7 bit addresses throughout. If you have a datasheet or sample code that uses 8 bit address, you'll want to drop the low bit (i.e. shift the value one bit to the right), yielding an address between 0 and 127. However the addresses from 0 to 7 are not used because are reserved so the first address that can be used is 8. Please note that a pull-up resistor is needed when connecting SDA/SCL pins. Please refer to the examples for more informations. MEGA 2560 board has pull-up resistors on pins 20 - 21 onboard.

The Wire library implementation uses a 32 byte buffer, therefore any communication should be within this limit. Exceeding bytes in a single transmission will just be dropped.

Examples

- Digital Potentiometer: Control an Analog Devices AD5171 Digital Potentiometer.
- Master Reader/Slave Writer: Program two Arduino boards to communicate with one another in a Master Reader/Slave Sender configuration via the I2C.
- Master Writer/Slave receiver:Program two Arduino boards to communicate with one another in a Master Writer/Slave Receiver configuration via the I2C.
- SFR Ranger Reader: Read an ultra-sonic range finder interfaced via the I2C.
- Add SerCom : Adding mores Serial interfaces to SAMD microcontrollers.

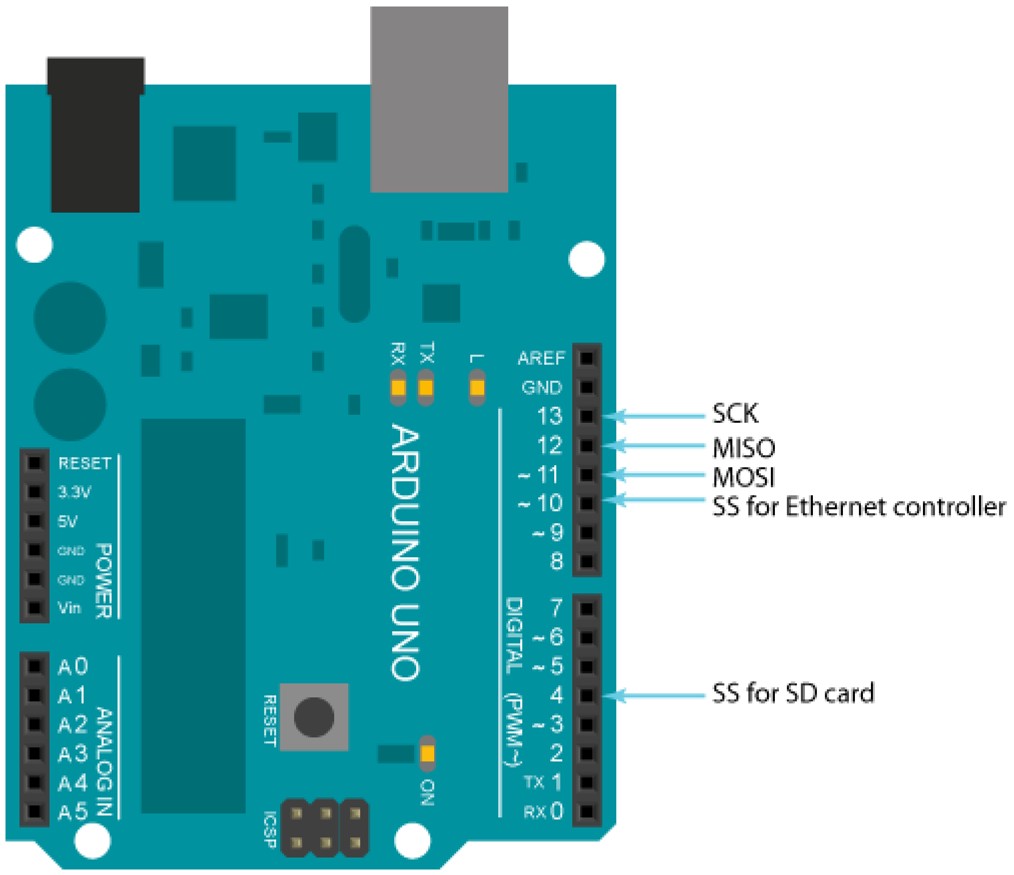
See also

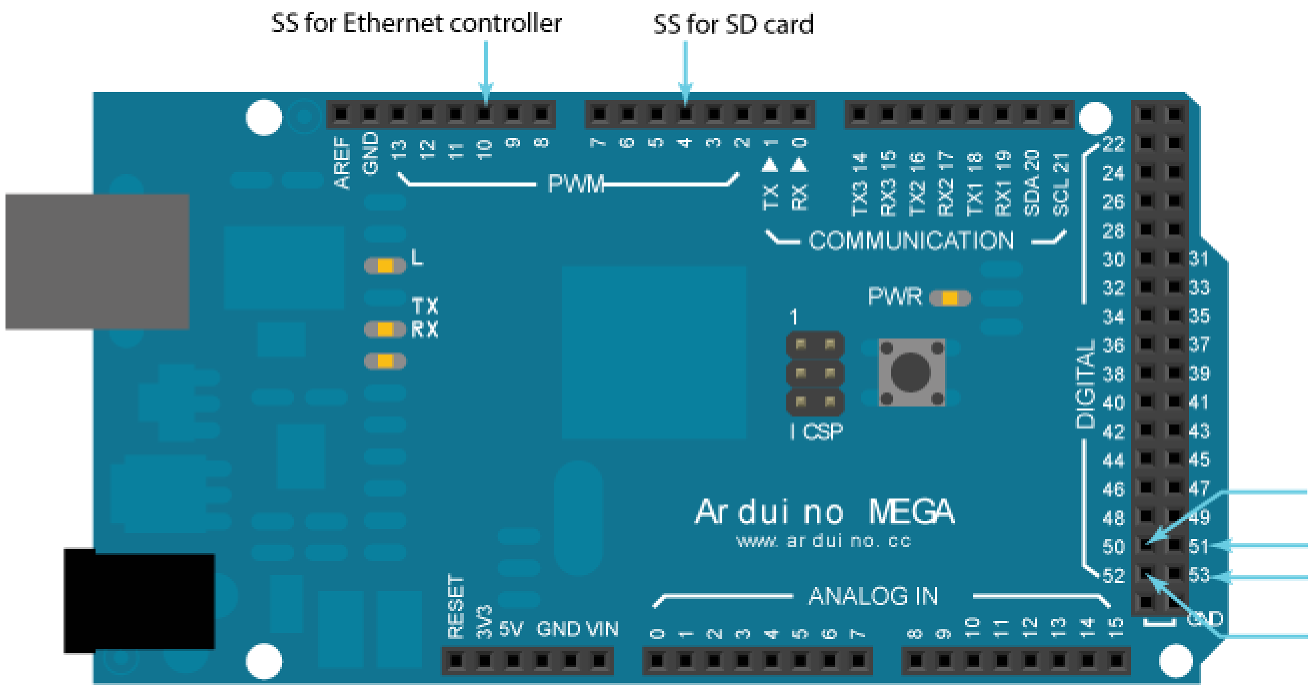
- Master Writer
- Master Reader
- SFR Ranger Reader
- Digital Potentiometer

Ethernet / Ethernet 2 library

These libraries are designed to work with the Arduino Ethernet Shield (Ethernet.h) or the Arduino Ethernet Shield 2 and Leonardo Ethernet (Ethernet2.h). The libraries are allow an Arduino board to connect to the internet. The board can serve as either a server accepting incoming connections or a client making outgoing ones. The libraries support up to four concurrent connection (incoming or outgoing or a combination). Ethernet library (Ethernet.h) manages the W5100 chip, while Ethernet2 library (Ethernet2.h) manages the W5500 chip; all the functions remain the same. Changing the library used allows to port the same code from Arduino Ethernet Shield to Arduino Ethernet 2 Shield or Arduino Leonardo Ethernet and vice versa.

Arduino communicates with the shield using the SPI bus. This is on digital pins 11, 12, and 13 on the Uno and pins 50, 51, and 52 on the Mega. **On both boards, pin 10 is used as SS.** On the Mega, the hardware SS pin, 53, is not used to select the W5100, but it must be kept as an output or the SPI interface won't work.





Examples

- ChatServer: set up a simple chat server.
- WebClient: make a HTTP request.
- WebClientRepeating: Make repeated HTTP requests.
- WebServer: host a simple HTML page that displays analog sensor values.
- BarometricPressureWebServer: outputs the values from a barometric pressure sensor as a web page.
- UDPSendReceiveString: Send and receive text strings via UDP.
- UdpNtpClient: Query a Network Time Protocol (NTP) server using UDP.
- DnsWebClient: DNS and DHCP-based Web client.

- DhcpChatServer: A simple DHCP Chat Server
- DhcpAddressPrinter: Get an IP address via DHCP and print it out
- TelnetClient: A simple Telnet client

SD Library

The SD library allows for reading from and writing to SD cards, e.g. on the Arduino Ethernet Shield. It is built on sdfatlib by William Greiman. The library supports FAT16 and FAT32 file systems on standard SD cards and SDHC cards. It uses short 8.3 names for files. The file names passed to the SD library functions can include paths separated by forward-slashes, /, e.g. "directory/filename.txt". Because the working directory is always the root of the SD card, a name refers to the same file whether or not it includes a leading slash (e.g. "/file.txt" is equivalent to "file.txt"). As of version 1.0, the library supports opening multiple files.

The communication between the microcontroller and the SD card uses SPI, which takes place on digital pins 11, 12, and 13 (on most Arduino boards) or 50, 51, and 52 (Arduino Mega). Additionally, another pin must be used to select the SD card. This can be the hardware SS pin - pin 10 (on most Arduino boards) or pin 53 (on the Mega) - or another pin specified in the call to SD.begin(). **Note that even if you don't use the hardware SS pin, it must be left as an output or the SD library won't work.**

Notes on using the Library and various shields

Examples

- Card Info: Get info about your SD card.
- Datalogger: Log data from three analog sensors to an SD card.
- Dump File: Read a file from the SD card.
- Files: Create and destroy an SD card file.
- List Files: Print out the files in a directory on a SD card.
- Read Write: Read and write data to and from an SD card.

Functions

stream.available()

Description

`available()` gets the number of bytes available in the stream. This is only for bytes that have already arrived.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

```
stream.available()
```

Parameters

`stream` : an instance of a class that inherits from Stream.

Returns

`int` : the number of bytes available to read

stream.read()

Description

`read()` reads characters from an incoming stream to the buffer.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the stream class main page for more information.

Syntax

`stream.read()`

Parameters

`stream` : an instance of a class that inherits from Stream.

Returns

The first byte of incoming data available (or -1 if no data is available).

stream.flush()

Description

`flush()` clears the buffer once all outgoing characters have been sent.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the stream class main page for more information.

Syntax

`stream.flush()`

Parameters

`stream` : an instance of a class that inherits from Stream.

Returns

`boolean`

stream.find()

Description

`find()` reads data from the stream until the target string of given length is found The function returns true if target string is found, false if timed out.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the stream class main page for more information.

Syntax

```
stream.find(target)
```

Parameters

`stream` : an instance of a class that inherits from Stream.

`target` : the string to search for (char)

Returns

```
boolean
```

stream.findUntil()

Description

`findUntil()` reads data from the stream until the target string of given length or terminator string is found.

The function returns true if target string is found, false if timed out

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the LANGUAGE Stream class main page for more information.

Syntax

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

Parameters

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

Returns

```
Boolean
```

stream.peek()

Description

Read a byte from the file without advancing to the next one. That is, successive calls to `peek()` will return the same value, as will the next call to `read()`.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

```
stream.peek()
```

Parameters

`stream` : an instance of a class that inherits from Stream.

Returns

The next byte (or character), or -1 if none is available.

stream.readBytes()

Description

`readBytes()` read characters from a stream into a buffer. The function terminates if the determined length has been read, or it times out (see setTimeout()).

`readBytes()` returns the number of bytes placed in the buffer. A 0 means no valid data was found.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

```
stream.readBytes(buffer, length)
```

Parameters

`stream` : an instance of a class that inherits from Stream.

`buffer` : the buffer to store the bytes in (`char[]` or `byte[]`)

`length` : the number of bytes to `read(int)`

Returns

The number of bytes placed in the buffer (`size_t`)

stream.readBytesUntil()

Description

`readBytesUntil()` reads characters from a stream into a buffer. The function terminates if the terminator character is detected, the determined length has been read, or it times out (see setTimeout()).

`readBytesUntil()` returns the number of bytes placed in the buffer. A 0 means no valid data was found.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

```
stream.readBytesUntil(character, buffer, length)
```

Parameters

`stream` : an instance of a class that inherits from Stream.
`character` : the character to search for (`char`)
`buffer`: the buffer to store the bytes in (`char[]` or `byte[]`)
`length : the number of bytes to `read(int)`

Returns

The number of bytes placed in the buffer.

stream.readString()

Description

`readString()` reads characters from a stream into a String. The function terminates if it times out (see setTimeout()).

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

```
stream.readString()
```

Parameters

Nothing

Returns

A String read from a stream.

stream.readStringUntil()

Description

`readStringUntil()` reads characters from a stream into a String. The function terminates if the terminator character is detected or it times out (see setTimeout()).

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

```
stream.readString(terminator)
```

Parameters

`terminator` : the character to search for (`char`)

Returns

The entire String read from a stream, until the terminator character is detected.

stream.parseInt()

Description

`parseInt()` returns the first valid (long) integer number from the current position. Initial characters that are not integers (or the minus sign) are skipped.

In particular:

- Initial characters that are not digits or a minus sign, are skipped;
- Parsing stops when no characters have been read for a configurable time-out value, or a non-digit is read;
- If no valid digits were read when the time-out (see Stream.setTimeout()) occurs, 0 is returned;

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

```
stream.parseInt(list)
stream.parseInt(''list', char skipchar')
```

Parameters

`stream` : an instance of a class that inherits from Stream.
`list` : the stream to check for ints (`char`)
`skipChar`: used to skip the indicated char in the search. Used for example to skip thousands divider.

Returns

`long`

stream.parseFloat()

Description

`parseFloat()` returns the first valid floating point number from the current position. Initial characters that are not digits (or the minus sign) are skipped. `parseFloat()` is terminated by the first character that is not a floating point number.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more informatio

Syntax

```
stream.parseFloat(list)
```

Parameters

`stream` : an instance of a class that inherits from Stream.

`list` : the stream to check for floats (`char`)

Returns

`float`

stream.setTimeout()

Description

`setTimeout()` sets the maximum milliseconds to wait for stream data, it defaults to 1000 milliseconds. This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the LANGUAGE Stream class main page for more information.

Syntax

```
stream.setTimeout(time)
```

Parameters

`stream` : an instance of a class that inherits from Stream. `time` : timeout duration in milliseconds (`long`).

Returns

Nothing

USB (32u4 based boards and Due/Zero only)

Keyboard [USB]

Description

The keyboard functions enable 32u4 or SAMD micro based boards to send keystrokes to an attached computer through their micro’s native USB port.

Note: Not every possible ASCII character, particularly the non-printing ones, can be sent with the Keyboard library.
The library supports the use of modifier keys. Modifier keys change the behavior of another key when pressed simultaneously. See here for additional information on supported keys and their use.

Notes and Warnings

These core libraries allow the 32u4 and SAMD based boards (Leonardo, Esplora, Zero, Due and MKR Family) to appear as a native Mouse and/or Keyboard to a connected computer.

A word of caution on using the Mouse and Keyboard libraries: if the Mouse or Keyboard library is constantly running, it will be difficult to program your board. Functions such as `Mouse.move()` and `Keyboard.print()` will move your cursor or send keystrokes to a connected computer and should only be called when you are ready to handle them. It is recommended to use a control system to turn this functionality on, like a physical switch or only responding to specific input you can control.

When using the Mouse or Keyboard library, it may be best to test your output first using `Serial.print()`. This way, you can be sure you know what values are being reported. Refer to the Mouse and Keyboard examples for some ways to handle this.

Functions

Keyboard.begin()

Description

When used with a Leonardo or Due board, `Keyboard.begin()` starts emulating a keyboard connected to a computer. To end control, use Keyboard.end().

Syntax

```
Keyboard.begin()
```

Parameters

Nothing

Returns

Nothing

Example Code

```
#include <Keyboard.h>

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //if the button is pressed
  if(digitalRead(2)==LOW){
    //Send the message
    Keyboard.print("Hello!");
  }
}
```

Keyboard.end()

Description

Stops the keyboard emulation to a connected computer. To start keyboard emulation, use Keyboard.begin().

Syntax

```
Keyboard.end()
```

Parameters

Nothing

Returns

Nothing

Example Code

```
#include <Keyboard.h>

void setup() {
  //start keyboard communication
  Keyboard.begin();
  //send a keystroke
  Keyboard.print("Hello!");
  //end keyboard communication
  Keyboard.end();
}

void loop() {
 //do nothing
}
```

Keyboard.press()

Description

When called, `Keyboard.press()` functions as if a key were pressed and held on your keyboard. Useful when using modifier keys. To end the key press, use Keyboard.release() or Keyboard.releaseAll().

It is necessary to call Keyboard.begin() before using `press()`.

Syntax

`Keyboard.press()`

Parameters

`char` : the key to press

Returns

`size_t` : number of key presses sent.

Example Code

```
#include <Keyboard.h>

// use this option for OSX:
char ctrlKey = KEY_LEFT_GUI;
// use this option for Windows and Linux:
//  char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
    // do nothing until pin 2 goes low
    delay(500);
  }
  delay(1000);
  // new document:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.releaseAll();
  // wait for new window to open:
  delay(1000);
```

Keyboard.print()

Description

Sends a keystroke to a connected computer.

`Keyboard.print()` must be called after initiating Keyboard.begin().

Syntax

```
Keyboard.print(character)
Keyboard.print(characters)
```

Parameters

`character` : a char or int to be sent to the computer as a keystroke characters : a string to be sent to the computer as a keystroke.

Returns

`size_t` : number of bytes sent.

Example Code

```
#include <Keyboard.h>

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //if the button is pressed
  if(digitalRead(2)==LOW){
    //Send the message
    Keyboard.print("Hello!");
  }
}
```

Notes and Warnings

When you use the `Keyboard.print()` command, the Arduino takes over your keyboard! Make sure you have control before you use the command. A pushbutton to toggle the keyboard control state is effective.

Keyboard.println()

Description

Sends a keystroke to a connected computer, followed by a newline and carriage return.

`Keyboard.println()` must be called after initiating Keyboard.begin().

Syntax

`Keyboard.println()`
`Keyboard.println(character)` + `Keyboard.println(characters)`

Parameters

`character` : a char or int to be sent to the computer as a keystroke, followed by newline and carriage return.

`characters` : a string to be sent to the computer as a keystroke, followed by a newline and carriage return.

Returns

`size_t` : number of bytes sent

Example Code

```
#include <Keyboard.h>

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //if the button is pressed
  if(digitalRead(2)==LOW){
    //Send the message
    Keyboard.println("Hello!");
  }
}
```

Notes and Warnings

When you use the Keyboard.println() command, the Arduino takes over your keyboard! Make sure you have control before you use the command. A pushbutton to toggle the keyboard control state is effective.

Keyboard.release()

Description

Lets go of the specified key. See Keyboard.press() for more information.

Syntax

`Keyboard.release(key)`

Parameters

`key` : the key to release. `char`

Returns

`size_t` : the number of keys released

Example Code

```
#include <Keyboard.h>

// use this option for OSX:
char ctrlKey = KEY_LEFT_GUI;
// use this option for Windows and Linux:
//  char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
    // do nothing until pin 2 goes low
    delay(500);
  }
  delay(1000);
  // new document:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.release(ctrlKey);
  Keyboard.release('n');
  // wait for new window to open:
  delay(1000);
}
```

Keyboard.releaseAll()

Description

Lets go of all keys currently pressed. See Keyboard.press() for additional information.

Syntax

`Keyboard.releaseAll()`

Parameters

Nothing

Returns

Nothing

Example Code

```
#include <Keyboard.h>

// use this option for OSX:
char ctrlKey = KEY_LEFT_GUI;
// use this option for Windows and Linux:
//  char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
    // do nothing until pin 2 goes low
    delay(500);
  }
  delay(1000);
  // new document:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.releaseAll();
  // wait for new window to open:
  delay(1000);
}
```

Keyboard.write()

Description

Sends a keystroke to a connected computer. This is similar to pressing and releasing a key on your keyboard. You can send some ASCII characters or the additional keyboard modifiers and special keys.

Only ASCII characters that are on the keyboard are supported. For example, ASCII 8 (backspace) would work, but ASCII 25 (Substitution) would not. When sending capital letters, Keyboard.write()

sends a shift command plus the desired character, just as if typing on a keyboard. If sending a numeric type, it sends it as an ASCII character (ex. Keyboard.write(97) will send 'a').

For a complete list of ASCII characters, see ASCIITable.com.

Syntax

```
Keyboard.write(character)
```

Parameters

`character` : a char or int to be sent to the computer. Can be sent in any notation that’s acceptable for a char. For example, all of the below are acceptable and send the same value, 65 or ASCII A:

```
Keyboard.write(65);         // sends ASCII value 65, or A
Keyboard.write('A');            // same thing as a quoted character
Keyboard.write(0x41);       // same thing in hexadecimal
Keyboard.write(0b01000001); // same thing in binary (weird choice, but it works)
```

Returns

`size_t` : number of bytes sent.

Example Code

```
#include <Keyboard.h>

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //if the button is pressed
  if(digitalRead(2)==LOW){
    //Send an ASCII 'A',
    Keyboard.write(65);
  }
}
```

Notes and Warnings

When you use the Keyboard.write() command, the Arduino takes over your keyboard! Make sure you have control before you use the command. A pushbutton to toggle the keyboard control state is effective.

Mouse

Description

The mouse functions enable 32u4 or SAMD micro based boards to control cursor movement on a connected computer through their micro's native USB port. When updating the cursor position, it is always relative to the cursor's previous location.

Notes and Warnings

These core libraries allow the 32u4 and SAMD based boards (Leonardo, Esplora, Zero, Due and MKR Family) to appear as a native Mouse and/or Keyboard to a connected computer.

A word of caution on using the Mouse and Keyboard libraries: if the Mouse or Keyboard library is constantly running, it will be difficult to program your board. Functions such as `Mouse.move()` and `Keyboard.print()` will move your cursor or send keystrokes to a connected computer and should only be called when you are ready to handle them. It is recommended to use a control system to turn this functionality on, like a physical switch or only responding to specific input you can control.

When using the Mouse or Keyboard library, it may be best to test your output first using `Serial.print()`. This way, you can be sure you know what values are being reported. Refer to the Mouse and Keyboard examples for some ways to handle this.

Functions

Mouse.begin()

Description

Begins emulating the mouse connected to a computer. `begin()` must be called before controlling the computer. To end control, use Mouse.end().

Syntax

`Mouse.begin()`

Parameters

Nothing

Returns

Nothing

Example Code

```
#include <Mouse.h>

void setup(){
 pinMode(2, INPUT);
}

void loop(){
  //initiate the Mouse library when button is pressed
  if(digitalRead(2) == HIGH){
     Mouse.begin();
   }

}
```

Mouse.click()

Description

Sends a momentary click to the computer at the location of the cursor. This is the same as pressing and immediately releasing the mouse button.

`Mouse.click()` defaults to the left mouse button.

Syntax

```
Mouse.click();
Mouse.click(button);
```

Parameters

`button`: which mouse button to press - `char`

- `MOUSE_LEFT` (default)
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Returns

Nothing

Example Code

```
#include <Mouse.h>

void setup(){
  pinMode(2,INPUT);
  //initiate the Mouse library
  Mouse.begin();
}

void loop(){
  //if the button is pressed, send a left mouse click
  if(digitalRead(2) == HIGH){
    Mouse.click();
  }
}
```

Notes and Warnings

When you use the `Mouse.click()` command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

Mouse.end()

Description

Stops emulating the mouse connected to a computer. To start control, use Mouse.begin().

Syntax

`Mouse.end()`

Parameters

Nothing

Returns

Nothing

Example Code

```
#include <Mouse.h>

void setup(){
  pinMode(2,INPUT);
  //initiate the Mouse library
  Mouse.begin();
}

void loop(){
  //if the button is pressed, send a left mouse click
  //then end the Mouse emulation
  if(digitalRead(2) == HIGH){
    Mouse.click();
    Mouse.end();
  }
}
```

Mouse.move()

Description

Moves the cursor on a connected computer. The motion onscreen is always relative to the cursor's current location. Before using `Mouse.move()` you must call Mouse.begin()

Syntax

`Mouse.move(xVal, yPos, wheel);`

Parameters

`xVal`: amount to move along the x-axis - `signed char`
`yVal`: amount to move along the y-axis - `signed char`
`wheel`: amount to move scroll wheel - `signed char`

Returns

Nothing

Example Code

```
#include <Mouse.h>

const int xAxis = A1;          //analog sensor for X axis
const int yAxis = A2;          // analog sensor for Y axis

int range = 12;                // output range of X or Y movement
int responseDelay = 2;        // response delay of the mouse, in ms
int threshold = range/4;       // resting threshold
int center = range/2;          // resting position value
int minima[] = {
  1023, 1023};                 // actual analogRead minima for {x, y}
int maxima[] = {
  0,0};                        // actual analogRead maxima for {x, y}
int axis[] = {
  xAxis, yAxis};               // pin numbers for {x, y}
int mouseReading[2];           // final mouse readings for {x, y}


void setup() {
 Mouse.begin();
}

void loop() {

// read and scale the two axes:
  int xReading = readAxis(0);
  int yReading = readAxis(1);

// move the mouse:
    Mouse.move(xReading, yReading, 0);
    delay(responseDelay);
}

/*
  reads an axis (0 or 1 for x or y) and scales the
  analog input range to a range from 0 to <range>
*/

int readAxis(int axisNumber) {
  int distance = 0;    // distance from center of the output range

  // read the analog input:
  int reading = analogRead(axis[axisNumber]);
// of the current reading exceeds the max or min for this axis,
// reset the max or min:
  if (reading < minima[axisNumber]) {
    minima[axisNumber] = reading;
  }
  if (reading > maxima[axisNumber]) {
    maxima[axisNumber] = reading;
  }

  // map the reading from the analog input range to the output range:
  reading = map(reading, minima[axisNumber], maxima[axisNumber], 0, range);

 // if the output reading is outside from the
 // rest position threshold,  use it:
  if (abs(reading - center) > threshold) {
    distance = (reading - center);
  }

  // the Y axis needs to be inverted in order to
  // map the movemment correctly:
  if (axisNumber == 1) {
    distance = -distance;
  }

  // return the distance for this axis:
  return distance;
}
```

Notes and Warnings

When you use the `Mouse.move()` command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

Mouse.press()

Description

Sends a button press to a connected computer. A press is the equivalent of clicking and continuously holding the mouse button. A press is cancelled with Mouse.release().

Before using `Mouse.press()`, you need to start communication with Mouse.begin().

`Mouse.press()` defaults to a left button press.

Syntax

```
Mouse.press();
Mouse.press(button)
```

Parameters

`button`: which mouse button to press - `char`

- `MOUSE_LEFT (default)`
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Returns

Nothing

Example Code

```
#include <Mouse.h>

void setup(){
  //The switch that will initiate the Mouse press
  pinMode(2,INPUT);
  //The switch that will terminate the Mouse press
  pinMode(3,INPUT);
  //initiate the Mouse library
  Mouse.begin();
}

void loop(){
  //if the switch attached to pin 2 is closed, press and hold the left mouse
button
  if(digitalRead(2) == HIGH){
    Mouse.press();
  }
  //if the switch attached to pin 3 is closed, release the left mouse button
  if(digitalRead(3) == HIGH){
    Mouse.release();
  }
}
```

Notes and Warnings

When you use the `Mouse.press()` command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

Mouse.release()

Description

Sends a message that a previously pressed button (invoked through Mouse.press()) is released. Mouse.release() defaults to the left button.

Syntax

```
Mouse.release();
Mouse.release(button);
```

Parameters

`button`: which mouse button to press - char

- `MOUSE_LEFT` (default)
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Returns

Nothing

Example Code

```
#include <Mouse.h>

void setup(){
  //The switch that will initiate the Mouse press
  pinMode(2,INPUT);
  //The switch that will terminate the Mouse press
  pinMode(3,INPUT);
  //initiate the Mouse library
  Mouse.begin();
}

void loop(){
  //if the switch attached to pin 2 is closed, press and hold the left mouse
button
  if(digitalRead(2) == HIGH){
    Mouse.press();
  }
  //if the switch attached to pin 3 is closed, release the left mouse button
  if(digitalRead(3) == HIGH){
    Mouse.release();
  }
}
```

Notes and Warnings

When you use the `Mouse.release()` command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

Mouse.isPressed()

Description

Checks the current status of all mouse buttons, and reports if any are pressed or not.

Syntax

```
Mouse.isPressed();
Mouse.isPressed(button);
```

Parameters

When there is no value passed, it checks the status of the left mouse button.

`button`: which mouse button to check - `char`

- `MOUSE_LEFT (default)`
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Returns

`boolean` : reports whether a button is pressed or not.

Example Code

```
#include <Mouse.h>

void setup(){
  //The switch that will initiate the Mouse press
  pinMode(2,INPUT);
  //The switch that will terminate the Mouse press
  pinMode(3,INPUT);
  //Start serial communication with the computer
  Serial1.begin(9600);
  //initiate the Mouse library
  Mouse.begin();
}

void loop(){
  //a variable for checking the button's state
  int mouseState=0;
  //if the switch attached to pin 2 is closed, press and hold the left mouse
button and save the state ina  variable
  if(digitalRead(2) == HIGH){
    Mouse.press();
    mouseState=Mouse.isPressed();
  }
  //if the switch attached to pin 3 is closed, release the left mouse button and
save the state in a variable
  if(digitalRead(3) == HIGH){
    Mouse.release();
    mouseState=Mouse.isPressed();
  }
  //print out the current mouse button state
  Serial1.println(mouseState);
  delay(10);
}
```