



Arduino Language Reference

By: Paul A. Teseny



The text of the Arduino Language Reference is licensed under a Creative Commons Attribution-ShareAlike 3.0 License. Code samples in the reference are released into the public domain. This is not a unique work; much of this material comes directly from the Arduino website. The information in this booklet is typeset for printing. The body text is Garamond and the section headings, code samples, headers and footers are Gill Sans MT.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark. The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

The information in this booklet is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, the author shall not have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

Table of Contents

Arduino Language Reference	7
Structure	7
setup()	7
loop()	7
Control Structures	8
Comparison Operators:	9
if / else	9
for statements	10
switch / case statements	12
while loops	13
do – while	13
break	14
continue	14
return	14
goto	15
Further Syntax	16
; semicolon	16
{} Curly Braces	16
Comments	18
#define	18
#include	19
Arithmetic Operators	20
= assignment operator (single equal sign)	20
Addition, Subtraction, Multiplication, & Division Description	20
% (modulo)	21
Comparison Operators	22
if (conditional) and ==, !=, <, > (comparison operators)	22
Boolean Operators	23
Pointer Operators	24
& (reference) and * (dereference)	24
Bitwise Operators	24
Bitwise AND (&)	24
Bitwise OR ()	25
Bitwise XOR (^)	26
Bitwise NOT (~)	26
bitshift left (<<), bitshift right (>>)	27



Compound Operators	28
++ (increment) / -- (decrement)	28
Shorthand Operators	29
Compound bitwise AND (&=), compound bitwise OR (=)	29
Compound bitwise AND (&=)	29
Compound bitwise OR (=)	30
Variables	32
Constants	32
Logical Levels, true and false (Boolean Constants)	32
Pin Levels, HIGH and LOW	32
Digital Pins, INPUT and OUTPUT	33
Pins Configured as Inputs	33
Pins Configured as Outputs	33
Integer Constants	33
U & L formatters	34
floating point constants	34
Data Types	36
void	36
boolean	36
char	37
unsigned char	37
byte	37
int	38
unsigned int	38
word	39
long	39
unsigned long	40
float	41
double	41
String - char array	42
String – object	43
Arrays	44
Conversion	46
char()	46
byte()	46
int()	46
word()	47
long()	47
float()	47

Variable Scope & Qualifiers	48
Variable Scope	48
Static	49
volatile keyword	50
const keyword	51
Utilities	52
sizeof	52
Functions	53
Digital I/O	53
pinMode()	53
digitalWrite()	54
digitalRead()	55
Analog I/O	56
analogReference(type)	56
analogRead()	56
analogWrite()	57
Advanced I/O	59
tone()	59
noTone()	59
shiftOut()	60
pulseIn()	62
Time	63
millis()	63
micros()	64
delay()	65
delayMicroseconds()	66
Math	67
min(x, y)	67
max(x, y)	67
abs(x)	68
constrain(x, a, b)	68
map(value, fromLow, fromHigh, toLow, toHigh)	69
pow(base, exponent)	70
sqrt(x)	70
Trigonometry	71
sin(rad)	71
cos(rad)	71
tan(rad)	71



Random Numbers	72
randomSeed(seed)	72
random()	72
Bits and Bytes	74
lowByte()	74
highByte()	74
bitRead()	74
bitWrite()	75
bitSet()	75
bitClear()	75
bit()	76
External Interrupts	77
attachInterrupt(interrupt, function, mode)	77
detachInterrupt(interrupt)	78
Interrupts	79
interrupts()	79
noInterrupts()	79
Communication	80
Serial	80
begin()	80
end()	81
available()	82
read()	83
flush()	84
print()	85
println()	87
write()	88
Libraries	89
Official Libraries	89
Contributed Libraries	89
ASCII Table	90
Arduino Language Summary	91

Arduino Language Reference

The Arduino language is based on C and C++ languages. The C understood by the Arduino compiler is not standard C; rather, it is a robust subset of standard C. A few standard C features are missing, but, the absence of those features is not crippling by any means. Arduino C, is more than able to perform just about any task you can throw at it. The missing features can easily be worked around, albeit sometimes in a less elegant manner.

Arduino programs can be divided in three main parts: *structure*, *variables*, and *functions*.

Structure

An Arduino program has two parts:

```
void setup()
void loop()
```

Setup() is preparation, and loop() is execution. In the setup section, always at the top of your program, you would set pinModes, initialize serial communication, etc. The loop section is the code to be executed -- reading inputs, triggering outputs, etc.

setup()

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 power-up or reset of the Arduino board.

Example:

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

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

loop()

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:

```
int buttonPin = 3;

// setup initializes serial and the button pin
void setup()
{
  beginSerial(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)
    serialWrite('H');
  else
    serialWrite('L');
  delay(1000);
}
```

Control Structures

if (conditional) and ==, !=, <, > (comparison operators)

if, which is used in conjunction with a comparison operator, tests whether a certain condition has been reached, such as an input being above a certain number. The format for an **if** test is:

```
if (someVariable > 50)
{
  // do something here
}
```

This program tests to see if someVariable is greater than 50. If it is, the program takes a particular action. Put another way, if the statement in parentheses is true, the statements inside the brackets are run. If not, the program skips over the 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
```

The statements being evaluated inside the parentheses require the use of one or more operators:

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)
```

Warning

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. `if` can also be part of a branching control structure using the `if...else` construction.

if / else

if/else allows greater control over the flow of code than the basic `if` statement, by allowing multiple tests to be grouped together. For example, an analog input could be tested and one action taken if the input was less than 500, and another action taken if the input was 500 or greater. The code would look like this:

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

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: 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.



Example:

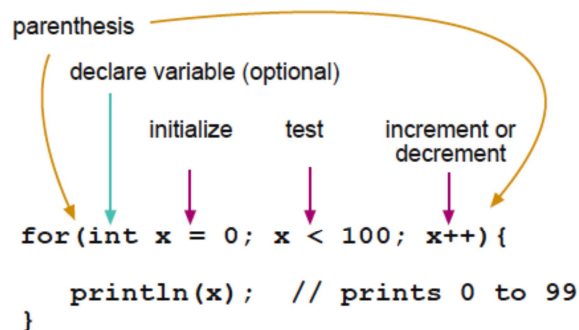
```
if (pinFiveInput < 500)
{
    // do Thing A
}
else if (pinFiveInput >= 1000)
{
    // do Thing B
}
else
{
    // do Thing C
}
```

Another way to express branching, mutually exclusive tests, is with the switch case statement.

for statements

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. There are three parts to the for loop header:

```
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:

```
void setup()
{
  // no setup needed
}

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

Coding Tips

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 statements

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 label:
    // statements
    break;
  case label:
    // statements
    break;
  default:
    // statements
}
```

Parameters:

var: the variable whose value to compare to the various cases
label: a value to compare the variable to

Example:

```
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
}
```

while loops

while loops 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(expression){  
  // statement(s)  
}
```

Parameters:

expression - a (boolean) C statement that evaluates to true or false

Example:

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

do – while

The do 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.

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

Example:

```
do  
{  
  delay(50);           // wait for sensors to stabilize  
  x = readSensors();  // check the sensors  
} while (x < 100);
```



break

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

Example:

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

continue

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

Example:

```
for (x = 0; x < 255; x ++)  
{  
    if (x > 40 && x < 120){           // create jump in values  
        continue;  
    }  
    digitalWrite(PWMPin, x);  
    delay(50);  
}
```

return

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

Examples:

```
// 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

Transfers program flow to a labeled point in the program

Syntax

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

Example:

```
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:
```

Tip

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 fully 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

; semicolon

Used to end a statement.

Example:

```
int a = 13;
```

Tip

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.

{ } Curly Braces

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.

At present this feature is slightly buggy as the IDE will often find (incorrectly) a brace in text that has been "commented out." Beginning 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.

Because the use of the curly brace is so varied, it is good programming practice to type the closing brace immediately after typing the opening brace when inserting a construct which requires curly braces. Then insert some carriage returns between your braces and begin inserting statements. Your braces, and your attitude, will never become unbalanced.

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.

The main uses of curly braces:

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)
}
```

Comments

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 on the processor.

Comments only purpose are to help you understand (or remember) how your program works or to inform others how your program works. There are two different ways of marking a line as a comment:

Example:

```
x = 5;           // This is a single line comment. Anything after the slashes is a comment
                // to the end of the line

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

Tip

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

#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**. Arduino defines have the same syntax as C defines:

Syntax:

```
#define constantName value    // Note that the # is necessary
```

Example:

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

Tip

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

#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.

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.

Example:

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

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

Arithmetic Operators

= assignment operator (single equal sign)

Stores the value to the right of the equal sign in the variable to the left of the equal sign.

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:

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

Programming Tips

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.

Don't confuse the assignment operator [=] (single equal sign) with the comparison operator [==] (double equal signs), which evaluates whether two expressions are equal.

Addition, Subtraction, Multiplication, & Division Description

These operators return the **sum**, **difference**, **product**, or **quotient** (respectively) of the two operands.

The operation is conducted using the data type of the operands, so, for example, 9 / 4 gives 2 since 9 and 4 are ints. This also means that the operation can overflow if the result is larger than that which can be stored in the data type (e.g. adding 1 to an int with the value 32,767 gives -32,768). If the operands are of different types, the "larger" type is used for the calculation.

If one of the numbers (operands) are of the type float or of type double, floating point math will be used for the calculation.

Examples:

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

Syntax:

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

Parameters:

```
value1: any variable or constant
value2: any variable or constant
```

Programming Tips

Know that integer constants default to int, so some constant calculations may overflow (e.g. $60 * 1000$ will yield a negative result).

Choose variable sizes that are large enough to hold the largest results from your calculations.

Know at what point your variable will "roll over" and also what happens in the other direction e.g. (0 - 1) OR (0 - - 32768).

For math that requires fractions, use float variables, but be aware of their drawbacks: large size, slow computation speeds.

Use the cast operator e.g. (int)myFloat to convert one variable type to another on the fly.

% (modulo)

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).

Syntax:

```
result = dividend % divisor
```

Parameters:

dividend: the number to be divided

divisor: the number to divide by

Returns (the remainder)

Examples:

```
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
```

Example Code:

```
/* 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
}
```

Tip

The modulo operator does not work on floats.

Comparison Operators

if (conditional) and ==, !=, <, > (comparison operators)

if, which is used in conjunction with a comparison operator, tests whether a certain condition has been reached, such as an input being above a certain number. The format for an if test is:

```
if (someVariable > 50)
{
    // do something here
}
```

The program tests to see if `someVariable` is greater than 50. **If** it is, the program takes a particular action. Put another way, if the statement in parentheses is true, the statements inside the brackets are run. **If** not, the program skips over the 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.

Examples:

```
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
```

The statements being evaluated inside the parentheses require the use of one or more operators:

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)
```

Warning

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.

`if` can also be part of a branching control structure using the `if...else` construction.

Boolean Operators

These can be used inside the condition of an `if` statement.

`&&` (logical and)

True only if both operands are true, e.g.

```
if (digitalRead(2) == HIGH && digitalRead(3) == HIGH) {           // read two switches
    // ...
}
```

is true only if both inputs are high.

`||` (logical or)

True if either operand is true, e.g.

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

is true if either `x` or `y` is greater than 0.

`!` (not)

True if the operand is false, e.g.

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

is true if `x` is false (i.e. if `x` equals 0).

Warning

Make sure you don't mistake the boolean AND operator, `&&` (double ampersand) for the bitwise AND operator `&` (single ampersand). They are entirely different beasts. Similarly, do not confuse the boolean `||` (double pipe) operator with the bitwise OR operator `|` (single pipe).

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.

Examples:

```
if (a >= 10 && a <= 20){           // true if a is between 10 and 20
```



Pointer Operators

& (reference) and * (dereference)

Pointers are one of the more 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

Bitwise AND (&)

The bitwise operators perform their calculations at the bit level of variables. They help solve a wide range of common programming problems.

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 | | operand1
0 1 0 | operand2
-----
0 0 0 | (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. 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

Bitwise OR (|)

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

```

Here is an example of the bitwise OR used in a snippet of C++ code:

```

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

```

Example Program:

A common job for the bitwise AND and OR operators is what programmers call Read-Modify-Write on a port. Writing to a port controls all of the pins at once.

PORTD is a built-in constant that refers to the output states of digital pins 0,1,2,3,4,5,6,7. If there is 1 in a bit position, then that pin is HIGH. If we write PORTD = B00110001; we have made pins 2,3 & 7 HIGH. One slight hitch here is that we may also have changed the state of Pins 0 & 1, which are used for serial communications so we may have interfered with serial communication.

Our algorithm for the program is:

Get PORTD and clear out only the bits corresponding to the pins we wish to control (with bitwise AND). Combine the modified PORTD value with the new value for the pins under control (with bitwise OR).

```

int i;    // counter variable
int j;
void setup(){
    // 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
    DDRD = DDRD | B11111100;
    Serial.begin(9600);
}
void loop(){
    for (i=0; i<64; i++){
        // clear out bits 2 - 7, leave pins 0 and 1 untouched (xx & 11 == xx)
        PORTD = PORTD & B00000011;
        j = (i << 2);           // shift variable up to pins 2 - 7 - to avoid pins 0 and 1
        PORTD = PORTD | j;      // combine the port information with the new information for LED pins
        Serial.println(PORTD, BIN); // debug to show masking
        delay(100);
    }
}

```



Bitwise XOR (^)

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 `^`. This operator is very similar to the bitwise OR operator `|`, only it evaluates to 0 for a given bit position when both of the input bits for that position are 1:

```

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

```

Another way to look at bitwise XOR is that each bit in the result is a 1 if the input bits are different, or 0 if they are the same.

Example:

```

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 OR 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 NOT (~)

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. For example:

```

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

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

```

Note

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 , $\sim x$ is the same as $-x-1$.

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

bitshift left (<<), bitshift right (>>)

There are two bit shift operators in C++: the left shift operator `<<` and the right shift operator `>>`. These operators cause the bits in the left operand to be shifted left or right by the number of positions specified by the right operand.

Syntax:

```
variable << number_of_bits
variable >> number_of_bits
```

Parameters:

```
variable - (byte, int, long) number_of_bits integer <= 32
```

Example:

```
int a = 5;           // binary: 0000000000000101
int b = a << 3;      // binary: 0000000000101000, or 40 in decimal
int c = b >> 3;      // binary: 0000000000000101, or back to 5 like we started with
```

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

```
int a = 5; // binary: 0000000000000101
int b = a << 14; // 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:

```
1 << 0 == 1
1 << 1 == 2
1 << 2 == 4
1 << 3 == 8
...
1 << 8 == 256
1 << 9 == 512
1 << 10 == 1024
...
```

When you shift x right by y bits ($x \gg y$), 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: 11111111110000
int y = x >> 3;       // binary: 11111111111110
```

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 ones being copied from the left:

```
int x = -16;           // binary: 11111111110000
int y = (unsigned int)x >> 3; // binary: 00011111111110
```

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

++ (increment) / -- (decrement)

Increment or decrement a variable

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
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 : an integer or long (possibly unsigned)

Returns:

The original or newly incremented / decremented value of the variable.

Examples:

```
x = 2;
y = ++x; // x now contains 3, y contains 3
y = x--; // x contains 2 again, y still contains 3
```

Shorthand Operators

Perform a mathematical operation (`+=`, `-=`, `*=`, `/=`) on a variable with another constant or variable. The `+=` (et al) operators are just a convenient shorthand for the expanded syntax, listed below.

Syntax:

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

Parameters

`x`: any variable type
`y`: any variable type or constant

Examples

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

Compound bitwise AND (&=), compound bitwise OR (|=)

The compound bitwise operators perform their calculations at the bit level of variables. They are often used to clear and set specific bits of a variable.

See the bitwise AND (&) and bitwise OR (|) operators for the details of their operation, and also the Bitmath Tutorial for more information on bitwise operators.

Compound bitwise AND (&=)

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.

Syntax:

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

Parameters:

`x`: a char, int or long variable
`y`: an integer constant or char, int, or long



Example:

First, 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

```

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;
```

Note

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 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 0 0 0 mask
-----
1 0 1 0 1 0 0 0 // variable 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 0 0 0 mask
-----
x x x x x x 0 0 // variable unchanged & bits cleared

```

So if:

```

myByte = 10101010;
myByte &= B11111000 == B10101000;

```

Compound bitwise OR (|=)

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

Syntax:

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

Parameters:

x: a char, int or long variable
y: an integer constant or char, int, or long

Example:

First, 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
```

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;
```

Consequently - to set bits 0 & 1 of a variable, while leaving the rest of the variable unchanged, use the compound bitwise AND 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 // variable unchanged & bits cleared
```

Here is the same representation with the variables bits replaced with the symbol 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 // variable unchanged & bits cleared
```

So if:

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

Variables

Constants

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

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

The true and false constants are typed in lowercase unlike HIGH, LOW, INPUT, & OUTPUT.

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 microcontroller will report HIGH if a voltage of 3 volts or more is present at the pin.

A pin may also be configured as an INPUT with pinMode, and subsequently made HIGH with digitalWrite, this will set the internal 20K pullup resistors, which will steer the input pin to a HIGH reading unless it is pulled LOW by external circuitry.

When a pin is configured to OUTPUT with pinMode, and set to HIGH with digitalWrite, the pin is at 5 volts. In this state it can source current, e.g. light an LED that is connected through a series resistor to ground, or to another pin configured as an output, and set to LOW.

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 microcontroller will report LOW if a voltage of 2 volts or less is present at the pin.

When a pin is configured to OUTPUT with pinMode, and set to LOW with digitalWrite, the pin is at 0 volts. In this state it can sink current, e.g. light an LED that is connected through a series resistor to, +5 volts, or to another pin configured as an output, and set to HIGH.

Digital Pins, INPUT and OUTPUT

Digital pins can be used either as INPUT or OUTPUT. Changing a pin from INPUT TO OUTPUT with `pinMode()` drastically changes the electrical behavior of the pin.

Pins Configured as Inputs

Arduino (Atmega) pins configured as INPUT with `pinMode()` are said to be in a high-impedance state. One way of explaining this is that pins configured as INPUT make extremely small demands on the circuit that they are sampling, say equivalent to a series resistor of 100 Megohms in front of the pin. This makes them useful for reading a sensor, but not powering an LED.

Pins Configured as Outputs

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 positive current) or sink (provide negative current) up to 40 mA (milliamps) of current to other devices/circuits. This makes them useful for powering LED's but useless for reading sensors. Pins configured as outputs can also be damaged or destroyed if short circuited to either ground or 5 volt power rails. The amount of current provided by an Atmega pin is also not enough to power most relays or motors, and some interface circuitry will be required.

Integer Constants

Integer constants are numbers used directly in a sketch, like 123. By default, these numbers are treated as `int`'s 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, AF, (a-f valid)

Decimal is 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:

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

Binary is base two. Only characters 0 and 1 are valid.

Example:

```
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 is base eight. Only characters 0 through 7 are valid. Octal values are indicated by the prefix "0"

Example:

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

Warning

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 (or hex) is base sixteen. 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 sited in upper or lower case (a-f).

Example:

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

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.

```
33u
```

a 'l' or 'L' to force the constant into a long data format.

```
100000L
```

a 'ul' or 'UL' to force the constant into an unsigned long constant.

```
32767ul
```

floating point constants

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.

Examples:

```
n = .005;
```

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

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:

```
// actions are performed in the functions "setup" and "loop"
// but no information is reported to the larger program
void setup()
{
    // ...
}

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

boolean

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

Example

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

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

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

unsigned char

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

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

byte

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

Example

```
byte b = B10010;           // "B" is the binary formatter (B10010 = 18 decimal)
```



int

Integers are your primary datatype for number storage, and store a 2 byte value. This yields a range of -32,768 to 32,767 (minimum value of -2^{15} and a maximum value of $(2^{15}) - 1$).

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.

Example:

```
int ledPin = 13;
```

Syntax:

```
int var = val;
```

var - your int variable name

val - the value you assign to that variable

Coding Tip

When variables are made to exceed their maximum capacity they "roll over" back to their minimum capacity, note that this happens in both directions.

```
int x
x = -32,768;
x = x - 1;           // x now contains 32,767 - rolls over in neg. direction
x = 32,767;
x = x + 1;           // x now contains -32,768 - rolls over
```

unsigned int

An **Unsigned int** (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 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.

Example:

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

Syntax:

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

Coding Tip

When variables are made to exceed their maximum capacity they "roll over" back to their minimum capacity, note that this happens in both directions

```
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
```

word

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

Example

```
word w = 10000;
```

long

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.

Example

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

Syntax

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

unsigned long

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 ($2^{32} - 1$).

Example:

```
// unsigned long time;
void setup()
{
  Serial.begin(9600);
}

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


float

The **float** datatype is 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 it 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.

Examples:

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

Syntax:

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

Example Code

```
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

Double precision floating point number. Occupies 4 bytes. The double implementation on the Arduino is currently exactly the same as the float, with no gain in precision.

Tip

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 the Arduino.

String - char array

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.

Examples:

```
// 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 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:

```
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 – object

The String class, part of the core as of version 0019, allows you to use and manipulate strings of text in more complex ways than character arrays do. You can concatenate Strings, append to them, search for and replace substrings, and more. It takes more memory than a simple character array, but it is also more useful.

For reference, character arrays are referred to as strings with a small **s**, and instances of the String class are referred to as Strings with a capital **S**. Note that constant strings, specified in "double quotes" are treated as char arrays, not instances of the String class.



Functions:

```
String()
charAt()
compareTo()
concat?()
endsWith()
equals()
equalsIgnoreCase()
getBytes()
indexOf()
lastIndexOf()
length()
replace?()
setCharAt()
startsWith()
substring()
toCharArray()
toLowerCase()
toUpperCase()
trim()
```

Operators:

```
[] (element access)
+ (concatenation)
== (comparison)
```

Arrays

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]);  
}
```



Conversion

char()

Converts a value to the char data type.

Syntax:

`char(x)`

Parameters:

`x`: a value of any type

Returns:

`char`

byte()

Converts a value to the byte data type.

Syntax:

`byte(x)`

Parameters:

`x`: a value of any type

Returns:

`Byte`

int()

Converts a value to the int data type.

Syntax:

`int(x)`

Parameters:

`x`: a value of any type

Returns:

`int`

word()

Convert a value to the word data type or create a word from two bytes.

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()

Converts a value to the long data type.

Syntax:

```
long(x)
```

Parameters:

x: a value of any type

Returns:

long

float()

Converts a value to the float data type.

Syntax:

```
float(x)
```

Parameters:

x: a value of any type

Returns:

float

Variable Scope & Qualifiers

Variable Scope

Variables in the C programming language, which Arduino uses, have a property called scope. This is in contrast to 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:

```
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

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

```

/* RandomWalk
 * 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()
{
  // tetst randomWalk function
  stepsize = 5;
  thisTime = randomWalk(stepsize);
  Serial.println(thisTime);
  delay(10);
}

int randomWalk(int moveSize){
  // 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
  static int place;
  place = place + (random(-moveSize, moveSize + 1));
  if (place < randomWalkLowRange){ // check lower and upper limits
    place = place + (randomWalkLowRange - place); // reflect
    number back in positive direction
  }
  else if(place > randomWalkHighRange){
    place = place - (place - randomWalkHighRange); // reflect
    number back in negative direction
  }
  return place;
}

```

volatile keyword

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.

Example:

```
// toggles LED when interrupt pin changes state
int pin = 13;
volatile int state = LOW;

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

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

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

const keyword

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

```
const float pi = 3.14;
float x;
// ...
x = pi * 2;           // it's fine to use const's in math
pi = 7;              // illegal - you can't write to (modify) a constant
```

#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

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

Syntax:

```
sizeof(variable)
```

Parameters:

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

Example code:

```
// 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.println(myStr[i], BYTE);
  }
}
```

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.

```
for (i = 0; i < (sizeof(myInts)/sizeof(int)) - 1; i++) {
  // do something with myInts[i]
}
```

Functions

Digital I/O

pinMode()

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

Syntax:

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

Parameters:

pin: the number of the pin whose mode you wish to set
 mode: either INPUT or OUTPUT

Returns:

None

Example:

```
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
}
```

Note

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

digitalWrite()

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, writing a HIGH value with `digitalWrite()` will enable an internal 20K pullup resistor (see the tutorial on digital pins). Writing LOW will disable the pullup. The pullup resistor is enough to light an LED dimly, so if LEDs appear to work, but very dimly, this is a likely cause. The remedy is to set the pin to an output with the `pinMode()` function.

Note

Digital pin 13 is harder to use as a digital input than the other digital pins because it has an LED and resistor attached to it that's soldered to the board on most boards. If you enable its internal 20k pull-up resistor, it will hang at around 1.7 V instead of the expected 5V because the onboard LED and series resistor pull the voltage level down, meaning it always returns LOW.

Syntax:

```
digitalWrite(pin, value)
```

Parameters:

pin: the pin number
value: HIGH or LOW

Returns:

none

Example:

```
// Sets pin 13 to HIGH, makes a one-second-long delay, and sets the pin back to LOW.
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
}
```

Note

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

digitalRead()

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 (int)

Returns:

HIGH or LOW

Example:

```
// Sets pin 13 to the same value as the pin 7, which is 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
}
```

Note

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

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

Analog I/O

analogReference(type)

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

- 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 ATmega328 and 2.56 volts on the ATmega8.
- EXTERNAL:** the voltage applied to the AREF pin is used as the reference.

Parameters:

type: which type of reference to use (DEFAULT, INTERNAL, or EXTERNAL).

Returns:

None.

Warning

If you're using an external reference voltage (applied to 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 micro-controller 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 $2.5 * 32 / (32 + 5) = \sim 2.2V$ at the AREF pin.

analogRead()

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:

```
int analogPin = 3;           // potentiometer wiper (middle terminal)
                             // outside leads to ground and +5V
connected to analog pin 3
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
}
```

Note

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()

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 is approximately 490 Hz.

On most Arduino boards (those with the ATmega168 or ATmega328), this function works on pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega, it works on pins 2 through 13. Older Arduino boards with an ATmega8 only support `analogWrite()` on pins 9, 10, and 11. You do not need to call `pinMode()` to set the pin as an output before calling `analogWrite()`.

The `analogWrite` function has nothing whatsoever to do with the analog pins or the `analogRead` function.



Syntax:

```
analogWrite(pin, value)
```

Parameters:

pin: the pin to write to.

value: the duty cycle: between 0 (always off) and 255 (always on).

Returns:

Nothing

Notes and Known Issues

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.

Example:

```
// 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
}
```

Advanced I/O

tone()

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.

Note

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.

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

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
duration: the duration of the tone in milliseconds (optional)

Returns:

nothing

noTone()

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

Note

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.

Syntax:

```
noTone(pin)
```

Parameters:

pin: the pin on which to stop generating the tone

Returns:

nothing



shiftOut()

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 to indicate that the bit is available.

This is a software implementation; Arduino (as of 0019) also provides an SPI library that uses the hardware implementation.

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:

None

Note

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(data, 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));
```

Example:

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);
  }
}

```

pulseIn()

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 uS 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:

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

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

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

Time

millis()

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.

Parameters:

None

Returns:

Number of milliseconds since the program started (unsigned long)

Example:

```
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
}
```

Tip

Note that the parameter for millis is an unsigned long, errors may be generated if a programmer tries to do math with other datatypes such as ints.



micros()

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.

Note

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

Parameters:

None

Returns:

Number of uS since the program started (unsigned long).

Example:

```
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
}
```


delay()

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:

```
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
}
```

Caveat

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()

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:

None

Example:

```
// configures pin number 8 to work as an output pin.
// It sends a train of pulses with 100 microseconds period.
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
}
```

Caveats and Known Issues

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(x, y)

Calculates the minimum of two numbers.

Parameters:

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

Returns:

The smaller of the two numbers.

Examples:

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

Note

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.

Warning

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

max(x, y)

Description

Calculates the maximum of two numbers.

Parameters:

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

Returns:

The larger of the two parameter values.

Example:

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

Note

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.

Warning

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

abs(x)

Computes the absolute value of a number.

Parameters:

x: the number

Returns:

x: if x is greater than or equal to 0.
-x: if x is less than 0.

Warning

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

constrain(x, a, b)

Constrains a number to be within a range.

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:

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

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

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. This 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

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.

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:

```
/* 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);
}
```

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(base, exponent)

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.

Parameters:

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

Returns:

The result of the exponentiation (double)

Example:

See the fscale function in the code library.

sqrt(x)

Calculates the square root of a number.

Parameters:

- x: the number, any data type

Returns:

The number's square root (double)

Trigonometry

sin(rad)

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

Parameters:

rad: the angle in radians (float)

Returns:

the sine of the angle (double)

cos(rad)

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

Parameters:

rad: the angle in radians (float)

Returns:

The cos of the angle ("double")

tan(rad)

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

Parameters:

rad: the angle in radians (float)

Returns:

The tangent of the angle (double)



Random Numbers

randomSeed(seed)

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:

long, int - pass a number to generate the seed.

Returns:

no returns

Example:

```
long randNumber;

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

void loop(){
  randNumber = random(300);
  Serial.println(randNumber);
  delay(50);
}
```

random()

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)

Note:

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.

Example:

```
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);
}
```

Bits and Bytes

lowByte()

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()

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()

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()

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 leastsignificant (rightmost) bit
- b: the value to write to the bit (0 or 1)

Returns:

none

bitSet()

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

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:

none

bitClear()

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:

none



bit()

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(interrupt, function, mode)

Specifies a function to call when an external interrupt occurs. Replaces any previous function that was attached to the interrupt. Most Arduino boards have two external interrupts: numbers 0 (on digital pin 2) and 1 (on digital pin 3). The Arduino Mega has an additional four: numbers 2 (pin 21), 3 (pin 20), 4 (pin 19), and 5 (pin 18).

Parameters:

interrupt: the number of the interrupt (int)

function: the function 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.

Returns:

none

Example:

```
int pin = 13;
volatile int state = LOW;

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

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

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



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.

Using Interrupts

Interrupts are useful for making things happen automatically in microcontroller programs, and can help solve timing problems. A good task for using an interrupt might be reading a rotary encoder, monitoring user input.

If you wanted to insure that a program always caught the pulses from a rotary encoder, never missing 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 doorbell.

`detachInterrupt(interrupt)`

Turns off the given interrupt.

Parameters:

`interrupt`: the number of interrupt to disable (0 or 1).

Interrupts

interrupts()

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.

Parameters:

None

Returns:

None

Example:

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

noInterrupts()

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.

Returns:

None

Example:

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

Communication

Serial

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()`.

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. (Don't connect these pins directly to an RS232 serial port; they operate at +/- 12V and can damage your Arduino board.)

begin()

Sets the data rate in bits per second (baud) for serial data transmission. For communicating with the computer, use one of these rates: 300, 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.

Syntax:

```
Serial.begin(speed)

// Arduino Mega only
Serial1.begin(speed)
Serial2.begin(speed)
Serial3.begin(speed)
```

Parameters:

speed: in bits per second (baud) – long

Returns:

nothing

Example:

```
void setup() {
  Serial.begin(9600); // opens serial port, sets data rate to 9600 bps
}
void loop() {}

// 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.

end()

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:

none

Returns:

nothing

available()

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 128 bytes).

Syntax:

```
Serial.available()

// Arduino Mega only
Serial1.available()
Serial2.available()
Serial3.available()
```

Parameters:

none

Returns:

The number of bytes available to read

Example:

```
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) {

    IncomingByte = Serial.read(); // read the incoming byte:

    Serial.print("I received: "); // say what you got
    Serial.println(incomingByte, DEC);
  }
}
```

Arduino Mega example:

```
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, BYTE);
  }
  // read from port 1, send to port 0:
  if (Serial1.available()) {
    int inByte = Serial1.read();
    Serial.print(inByte, BYTE);
  }
}
```

read()

Reads incoming serial data.

Syntax:

```
Serial.read()

// Arduino Mega only
Serial1.read()
Serial2.read()
Serial3.read()
```

Parameters:

None

Returns:

The first byte of incoming serial data available (or -1 if no data is available) – int

Example:

```
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) {

    incomingByte = Serial.read(); // read the incoming byte
    Serial.print("I received: "); // say what you got
    Serial.println(incomingByte, DEC);
  }
}
```

flush()

Flushes the buffer of incoming serial data. That is, any call to `Serial.read()` or `Serial.available()` will return only data received after all the most recent call to `Serial.flush()`.

Syntax:

```
Serial.flush()

// Arduino Mega only
Serial1.flush()
Serial2.flush()
Serial3.flush()
```

Parameters:

none

Returns:

nothing

print()

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(byte(78)) gives "N" (whose ASCII value is 78)  
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 BYTE, 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, BYTE) gives "N"  
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"
```

Syntax:

```
Serial.print(val)  
Serial.print(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:

None

Example:

```
/*
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.print("\t");
  Serial.println("BYTE");
  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"); // prints a tab
    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.print(x, BIN); // print as an ASCII-encoded binary
    Serial.print("\t"); // prints a tab
    Serial.println(x, BYTE); // prints as a raw byte value,
    // then adds the carriage return with "println"
    delay(200); // delay 200 milliseconds
  }
  Serial.println(""); // prints another carriage return
}
```

Programming Tip

The last character to be printed is transmitted over the serial port after `Serial.print()` has returned.

println()

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:

None

Example:

```
/*
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
  Serial.println(analogValue, BYTE); // print as a raw byte value
  // delay 10 milliseconds before the next reading:
  delay(10);
}
```



write()

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
`len`: the length of the buffer

Libraries

To use an existing library in a sketch, go to the Sketch menu, choose "Import Library", and pick from the libraries available. This will insert one or more **#include** statements at the top of the sketch and allow it to use the library.

Because libraries are uploaded to the board with your sketch, they increase the amount of space it takes up. If a sketch no longer needs a library, simply delete its **#include** statements from the top of your code.

Official Libraries

These are the "official" libraries that are included in the Arduino distribution.

EEPROM - reading and writing to "permanent" storage

SoftwareSerial - for serial communication on any digital pins

Stepper - for controlling stepper motors

Wire - Two Wire Interface (TWI/I2C) for sending and receiving data over a net of devices or sensors.

These libraries are compatible Wiring versions, and the links below point to the (excellent) Wiring documentation.

Matrix - Basic LED Matrix display manipulation library

Sprite - Basic image sprite manipulation library for use in animations with an LED matrix

Contributed Libraries

Libraries written by members of the Arduino community.

DateTime - a library for keeping track of the current date and time in software.

Firmata - for communicating with applications on the computer using a standard serial protocol.

GLCD - graphics routines for LCD based on the KS0108 or equivalent chipset.

LCD - control LCDs (using 8 data lines)

LCD 4 Bit - control LCDs (using 4 data lines)

LedControl - for controlling LED matrices or seven-segment displays with a MAX7221 or MAX7219.

LedControl - an alternative to the Matrix library for driving multiple LEDs with Maxim chips.

TextString - handle strings

Metro - help you time actions at regular intervals

MsTimer2 - uses the timer 2 interrupt to trigger an action every N milliseconds.

OneWire - control devices (from Dallas Semiconductor) that use the One Wire protocol.

PS2Keyboard - read characters from a PS2 keyboard.

Servo - provides software support for Servo motors on any pins.

Servotimer1 - provides hardware support for Servo motors on pins 9 and 10

Simple Message System - send messages between Arduino and the computer

SSerial2Mobile - send text messages or emails using a cell phone (via AT commands over software serial)

X10 - Sending X10 signals over AC power lines

To install, unzip the library to a sub-directory of the **hardware/libraries** sub-directory of the Arduino application directory. Then launch the Arduino environment; you should see the library in the **Import Library** menu.

ASCII Table

Dec	Hex	Char	Description	Dec	Hex	Char	Dec	Hex	Char
0	0		null	43	2B	+	86	56	V
1	1		start of heading	44	2C	,	87	57	W
2	2		start of text	45	2D	-	88	58	X
3	3		end of text	46	2E	.	89	59	Y
4	4		end of trans	47	2F	/	90	5A	Z
5	5		enquiry	48	30	0	91	5B	[
6	6		acknowledge	49	31	1	92	5C	\
7	7		bell	50	32	2	93	5D]
8	8		backspace	51	33	3	94	5E	^
9	9		horizontal tab	52	34	4	95	5F	_
10	A		new line	53	35	5	96	60	`
11	B		vertical tab	54	36	6	97	61	a
12	C		new page	55	37	7	98	62	b
13	D		carriage return	56	38	8	99	63	c
14	E		shift out	57	39	9	100	64	d
15	F		shift in	58	3A	:	101	65	e
16	10		data link escape	59	3B	;	102	66	f
17	11		device control 1	60	3C	<	103	67	g
18	12		device control 2	61	3D	=	104	68	h
19	13		device control 3	62	3E	>	105	69	i
20	14		device control 4	63	3F	?	106	6A	j
21	15		neg ack	64	40	@	107	6B	k
22	16		synchronous idle	65	41	A	108	6C	l
23	17		end of trans block	66	42	B	109	6D	m
24	18		cancel	67	43	C	110	6E	n
25	19		end of medium	68	44	D	111	6F	o
26	1A		substitute	69	45	E	112	70	p
27	1B		escape	70	46	F	113	71	q
28	1C		file separator	71	47	G	114	72	r
29	1D		group separator	72	48	H	115	73	s
30	1E		record separator	73	49	I	116	74	t
31	1F		unit separator	74	4A	J	117	75	u
32	20		space	75	4B	K	118	76	v
33	21	!		76	4C	L	119	77	w
34	22	"		77	4D	M	120	78	x
35	23	#		78	4E	N	121	79	y
36	24	\$		79	4F	O	122	7A	z
37	25	%		80	50	P	123	7B	{
38	26	&		81	51	Q	124	7C	
39	27	'		82	52	R	125	7D	}
40	28	(83	53	S	126	7E	~
41	29)		84	54	T	127	7F	DEL
42	2A	*		85	55	U			

Arduino Language Summary

Structure

In Arduino, the standard program entry point (main) is defined in the core and calls into two functions in a sketch. **setup()** is called once, then **loop()** is called repeatedly (until the board is reset).

```
void setup()
void loop()
```

Control Structures

```
if
if...else
for
switch case
while
do... while
break
continue
return
```

Further Syntax

```
; (semicolon)
{} (curly braces)
// (single line comment)
/**/ (multi-line comment)
#define
#include
```

Arithmetic Operators

```
+ (addition)
- (subtraction)
* (multiplication)
/ (division)
% (modulo)
```

Comparison Operators

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

Boolean Operators

```
&& (and)
|| (or)
! (not)
```

Bitwise Operators

```
& (bitwise and)
| (bitwise or)
^ (bitwise xor)
~ (bitwise not)
<< (bitshift left)
>> (bitshift right)
```

Compound Operators

```
++ (increment)
-- (decrement)
+= (compound addition)
-= (compound subtraction)
*= (compound multiplication)
/= (compound division)
&= (compound bitwise and)
|= (compound bitwise or)
```

Pointer Access Operators

```
* dereference operator
& reference operator
```

Variables

Constants

```
HIGH | LOW
INPUT | OUTPUT
true | false
integer constants
floating point constants
```

Data Types

```
void keyword
boolean
char
unsigned char
byte
int
unsigned int
long
unsigned long
float
double
string
array
```

Variable Scope & Qualifiers

```
static
volatile
const
PROGMEM
```

Functions

Math

```
min(x, y)
max(x, y)
abs(x)
constrain(x, a, b)
map(value, frL, frH, toL, toH)
pow(base, exponent)
sqrt(x)
```

Trigonometry

```
sin(rad)
cos(rad)
tan(rad)
```

Random Numbers

```
randomSeed(seed)
long random(max)
long random(min, max)
```

Digital I/O

```
pinMode(pin, mode)
digitalWrite(pin, value)
int digitalRead(pin)
```

Analog I/O

```
analogReference(type)
int analogRead(pin)
analogWrite(pin, value) – PWM
```

Advanced I/O

```
shiftOut(dPin, clkPin, bitOrd, val)
unsigned long pulseIn(pin, value)
```

Time

```
unsigned long millis()
delay(ms)
delayMicroseconds(us)
```

Port Manipulation

```
DDRX Data Direction Register
PORTX Data Register
PINX Input Pin - read only
X = B (digital pin 8 to 13)
C (analog input pins)
D (digital pins 0 to 7)
```

Serial Communication

```
Serial.begin(speed)
int Serial.available()
int Serial.read()
Serial.flush()
Serial.print(data)
Serial.println(data)
```

Interrupts

```
interrupts()
noInterrupts()
```

External Interrupts

```
attachInterrupt(int, func, mode)
detachInterrupt(interrupt)
```

