# Lesson 0 Language Reference

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

# Structure

## Structure

### 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 powerup 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

```
 const int buttonPin = 3;
// setup initializes serial and the button pin
void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
```

```
}
// loop checks the button pin each time,
// and will send serial if it is pressed
void loop()
{
   if (digitalRead(buttonPin) == HIGH)
      Serial.write('H');
   else
      Serial.write('L');
   delay(1000);
}
```

# Control Structures

## if

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.

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

```
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.

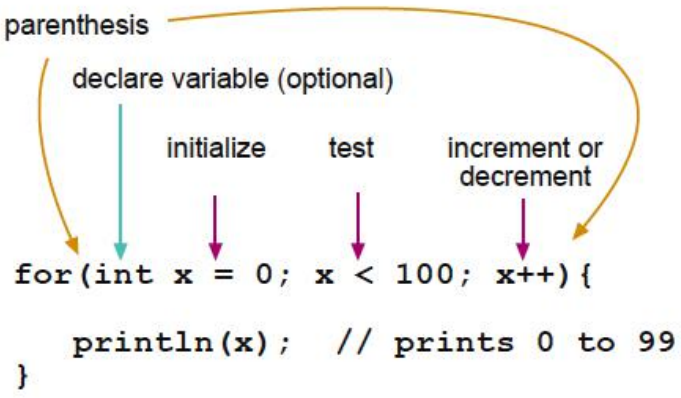
## for

### Description

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

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

```
// Dim an LED using a PWM pin
int PWMpin = 10; // LED in series with 470 ohm resistor on pin 10
void setup()
{
  // no setup needed
}
void loop()
{
    for (int i=0; i <= 255; i++){
```

```
        analogWrite(PWMpin, i);
        delay(10);
    }
}
```

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

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.

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
    break;
 }
```

Note

Please note that in order to declare variables within a case brackets are needed. An example is showed below.

```
switch (var) {
    case 1:
      {
      //do something when var equals 1
      int a = 0;
       .......
       .......
      }
       break;
    default:
      // if nothing else matches, do the default
      // default is optional
    break;
 }
```

Syntax

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

Parameters

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

label: a value to compare the variable to

## while

### Description

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 ++)
{
    analogWrite(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;
    }
```

```
  analogWrite(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

### 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 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.

### 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:
```

# 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 Atmega chip.

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.

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

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

### Example

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

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

prog_uint16_t myConstants[] PROGMEM = {0, 21140, 702    , 9128,    0, 25764, 8456,

0,0,0,0,0,0,0,0,29810,8968,29762,29762,4500};
```

# Arithmetic Operators

## = 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;

r = r / 5;
```

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)

Description

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

### == (equal to)

### != (not equal to)

### < (less than)

### > (greater than)

### <= (less than or equal to)

### >= (greater than or equal to)

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.

```
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 Access 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 (&), Bitwise OR (|), Bitwise XOR (^)

### Bitwise AND (&)

The bitwise operators perform their calculations at the bit level of variables. They help solve a wide range of common programming problems. Much of the material below is from an excellent tutorial on bitwise math which may be found here.

### Description and Syntax

Below are descriptions and syntax for all of the operators. Further details may be found in the referenced tutorial.

## Bitwise AND (&)

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

```
   0   0   1   1     operand1

   0   1   0   1     operand2

   ---------
```

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

In Arduino, the type int is a 16-bit value, so using & between two int expressions causes 16 simultaneous AND operations to occur. 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   1   1       operand1

0   1   0   1       operand2

----------

0   1   1   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 for Arduino Uno

A common job for the bitwise AND and OR operators is what programmers call Read-Modify-Write on a port. On microcontrollers, a port is an 8 bit number that represents something about the condition of the pins. 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 an bit position, then that pin is HIGH. (The pins already need to be set to outputs with the pinMode() command.) So if we write PORTD = B00110001; we have made pins 0,4 & 5 HIGH. One slight hitch here is that we may also have changeed the state of Pins 0 & 1, which are used by the Arduino 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 biwise OR).

```
int i;          // counter variable

int j;

void setup(){

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

// same as pinMode(pin, OUTPUT) for pins 2 to 7

Serial.begin(9600);

}

void loop(){

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

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

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   1   1     operand1

    0   1   0   1     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.

Here is a simple code 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   1       operand1

----------

 1   0     ~ operand1

 int a = 103;       // binary:    0000000001100111

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

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

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

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

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

### Description

From 聽 The Bitmath Tutorial 聽 in The Playground

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.

More on bitwise math may be found 聽 here.

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

```
int x = -16;         // binary: 1111111111110000

int y = x >> 3;    // binary: 1111111111111110
```

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

```
int x = -16;                              // binary: 1111111111110000

int y = (unsigned int)x >> 3;    // binary: 0001111111111110
```

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

```
int x = 1000;

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

# Compound Operators

## ++ (increment) / -- (decrement)

### Description

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
```

## += (compound addition)

## -= (compound subtraction)

## *= (compound multiplication)

## /= (compound division)

## %= (compound modulo)

### Description

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;

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

x %= 5;           // x now contains 0
```

# &= (compound bitwise and)

## Description

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

## 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 (hmmm something philosophical there?)

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

```
   1   0   1   0   1   0   1   0      variable

   1   1   1   1   1   1   0   0      mask
```

```
   ----------------------

   1   0   1   0   1   0   0   0

 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   1   0   0       mask

   ----------------------

   x   x   x   x   x   x   0   0

 variable unchanged

                            bits cleared
```

So if:

```
myByte =    B10101010;

myByte &= B11111100 == B10101000;
```

## |= (compound bitwise or)

### Description

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

### 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 聽 ORed" with 0 are unchanged, so if myByte is a byte variable,
myByte | B00000000 = myByte;

Bits that are "bitwise 聽 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 OR operator (|=) with the constant B00000011

```
  1   0   1   0   1   0   1   0     variable

  0   0   0   0   0   0   1   1     mask

  ----------------------

  1   0   1   0   1   0   1   1

variable unchanged

                          bits set
```

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

```
  x   x   x   x   x   x   x   x     variable

  0   0   0   0   0   0   1   1     mask

  ----------------------

  x   x   x   x   x   x   1   1
```

```
variable unchanged

                        bits set
```

So if:

```
myByte =

myByte |= B00000011 == B10101011;
```

# Variables

## Constants

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

### True and False

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

#### false

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

#### true

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

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

### HIGH and LOW

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

#### HIGH

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

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

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

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

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

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

### LOW

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

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

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

## INPUT, INPUT_PULLUP, and OUTPUT

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

### Pins Configured as INPUT

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

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

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

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

### Pins Configured as INPUT_PULLUP

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

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

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

### Pins Configured as 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 current) or sink (absorb current) up to 40 mA (milliamps) of current to other devices/circuits. This makes them useful for powering LEDs because LEDs typically use less than 40 mA. Loads greater than 40 mA (e.g. motors) will require a transistor or other interface circuitry.

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

## LED_BUILTIN

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

## Integer Constants

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

## floating point constants

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

### Description

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

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

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

Example

```
char myChar = 'A';

char myChar = 65;          // both are equival
```

## unsigned char

Description

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

The unsigned char datatype encodes numbers from 0 to 255.

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

Example

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

## byte

Description

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

Description

Integers are your primary data-type for number storage.

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

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

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

Example

```
int ledPin = 13;
```

Syntax

```
int var = val;
```

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

## unsigned int

Description

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

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

The difference between unsigned ints and (signed) ints, lies in the way the highest bit, sometimes refered 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 capacitiy, 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

### Description

On the Uno and other ATMEGA based boards, a word stores a 16-bit unsigned number. On the Due and Zero instead it stores a 32-bit unsigned number.

### Example

```
word w = 10000;
```

## long

### Description

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

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

### Example

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

### Syntax

```
long var = val;
```

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

## unsigned long

## Description

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

## Example

```
unsigned long time;



void setup()

{

  Serial.begin(9600);

}



void loop()

{

  Serial.print("Time: ");

  time = millis();

  //prints time since program started

  Serial.println(time);

  // wait a second so as not to send massive amounts of data

  delay(1000);

}
```

## Syntax

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

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

## short

### Description

A `short` is a 16-bit data-type.

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

### Example

```
short ledPin = 13;
```

### Syntax

```
short var = val;
```

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

## float

### Description

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

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

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

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

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

### 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

### Description

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

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

### 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 ATMEGA based Arduinos.

# string - char array

## Description

Text strings can be represented in two ways. you can use the String data type, which is part of the core as of version 0019, or you can make a string out of an array of type char and null-terminate it. This page described the latter method. For more details on the String object, which gives you more functionality at the cost of more memory, see the 聽 String object 聽 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 character. Str4 will be automatically

sized to eight characters, one for the extra null. In Str3, we've explicitly included the null character (written '\0') ourselves.

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

Single quotes or double quotes?

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

Wrapping long strings

You can wrap long strings like this:

```
char myString[] = "This is the first line"

" this is the second line"

" etcetera";
```

Arrays of strings

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

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

Example

```
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

Description

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.

Examples

## StringConstructors

The String object allows you to manipulate strings of text in a variety of useful ways. You can append characters to Strings, combine Strings through concatenation, get the length of a String, search and replace substrings, and more. This tutorial shows you how to initialize String objects.

```
String stringOne = "Hello String";                          // using a constant String

String stringOne =   String('a');                           // converting a constant char into a
String

String stringTwo =   String("This is a string");            // converting a constant string into a
String object

String stringOne =   String(stringTwo + " with more");      // concatenating two strings
```

```
String stringOne =  String(13);                 // using a constant integer

String stringOne =  String(analogRead(0), DEC);   // using an int and a base

String stringOne =  String(45, HEX);            // using an int and a base
(hexadecimal)

String stringOne =  String(255, BIN);           // using an int and a base (binary)

String stringOne =  String(millis(), DEC);      // using a long and a base

String stringOne =  String(5.698, 3);           // using a float and the decimal
places
```

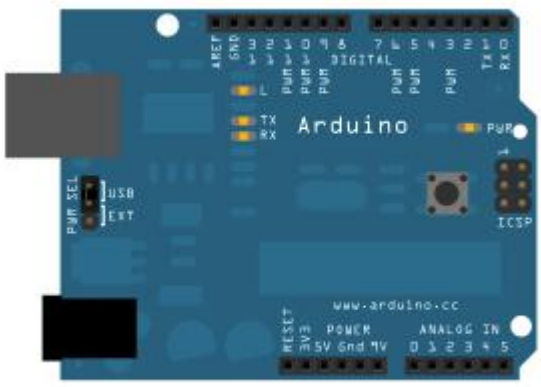
All of these methods are valid ways to declare a String object. They all result in an object containing a string of characters that can be manipulated using any of the String methods. To see them in action, upload the code below onto an Arduino or Genuino board and open the Arduino IDE serial monitor. You'll see the results of each declaration. Compare what's printed by each println() to the declaration above it.

## Hardware Required

Arduino or Genuino Board

## Circuit

There is no circuit for this example, though your board must be connected to your computer via USB and the serial monitor window of the Arduino Software (IDE) should be open.




## Code

```
/*
  String constructors

 Examples of how  to create strings from other data types
```

```
 created 27 July 2010
 modified 30 Aug 2011
 by Tom Igoe

 http://www.arduino.cc/en/Tutorial/StringConstructors

 This example code is in the public domain.
 */

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
 }

 // send an intro:
 Serial.println("\n\nString Constructors:");
 Serial.println();
}

void loop() {
 // using a constant String:
  String stringOne = "Hello String";
 Serial.println(stringOne);       // prints "Hello String"

 // converting a constant char into a String:
  stringOne =  String('a');
 Serial.println(stringOne);        // prints "a"

 // converting a constant string into a String object:
  String stringTwo =  String("This is a string");
 Serial.println(stringTwo);       // prints "This is a string"

 // concatenating two strings:
  stringOne =  String(stringTwo + " with more");
 // prints "This is a string with more":
 Serial.println(stringOne);

 // using a constant integer:
  stringOne =  String(13);
 Serial.println(stringOne);       // prints "13"
```

```
// using an int and a base:
 stringOne =  String(analogRead(A0), DEC);
// prints "453" or whatever the value of analogRead(A0) is
Serial.println(stringOne);

// using an int and a base (hexadecimal):
 stringOne =  String(45, HEX);
// prints "2d", which is the hexadecimal version of decimal 45:
Serial.println(stringOne);

// using an int and a base (binary)
 stringOne =  String(255, BIN);
// prints "11111111" which is the binary value of 255
Serial.println(stringOne);

// using a long and a base:
 stringOne =  String(millis(), DEC);
// prints "123456" or whatever the value of millis() is:
Serial.println(stringOne);

//using a float and the right decimal places:
 stringOne = String(5.698, 3);
Serial.println(stringOne);

//using a float and less decimal places to use rounding:
 stringOne = String(5.698, 2);
Serial.println(stringOne);

// do nothing while true:
while (true);

}
```

## StringAdditionOperator

You can add Strings together in a variety of ways. This is called concatenation and it results in the original String being longer by the length of the String or character array with which you concatenate it. The + operator allows you to combine a String with another String, with a constant character array, an ASCII representation of a constant or variable number, or a constant character.

```
// adding a constant integer to a string:

  stringThree =    stringOne + 123;
```

```
// adding a constant long interger to a string:

stringThree = stringOne + 123456789;

// adding a constant character to a string:

stringThree =    stringOne + 'A';

// adding a constant string to a string:

stringThree =    stringOne +    "abc";

// adding two Strings together:

stringThree = stringOne + stringTwo;
```

You can also use the + operator to add the results of a function to a String, if the function returns one of the allowed data types mentioned above. For example,

```
stringThree = stringOne + millis();
```

This is allowable since the millis() function returns a long integer, which can be added to a String. You could also do this:

```
stringThree = stringOne + analogRead(A0);
```

because analogRead() returns an integer. String concatenation can be very useful when you need to display a combination of values and the descriptions of those values into one String to display via serial communication, on an LCD display, over an Ethernet connection, or anywhere that Strings are useful.

Caution: You should be careful about concatenating multiple variable types on the same line, as you may get unexpected results. For example:

```
int sensorValue = analogRead(A0);

String stringOne = "Sensor value: ";

String stringThree = stringOne + sensorValue;

Serial.println(stringThree);
```

results in "Sensor Value: 402" or whatever the analogRead() result is, but

```
int sensorValue = analogRead(A0);

String stringThree = "Sensor value: " + sensorValue;

Serial.println(stringThree);
```

gives unpredictable results because `stringThree` never got an initial value before you started concatenating different data types.

Here's another example where improper initialization will cause errors:

```
Serial.println("I want " + analogRead(A0) + " donuts");
```

This won't compile because the compiler doesn't handle the operator precedence correctly. On the other hand, the following will compile, but it won't run as expected:

```
int sensorValue = analogRead(A0);

String stringThree = "I want " + sensorValue;

Serial.println(stringThree    + " donuts");    un correctly for the same reason as
```

before: stringThree never got an initial value before you started concatenating different data types.

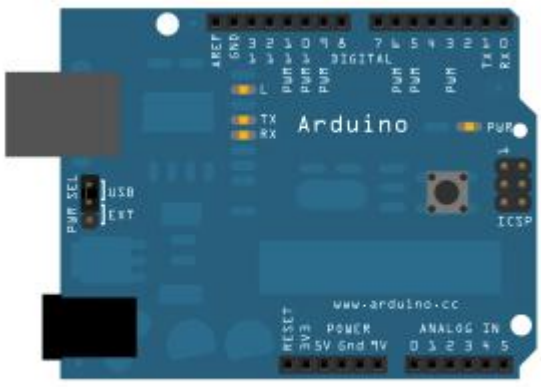
For best results, initialize your Strings before you concatenate them.

## Hardware Required

Arduino or Genuino Board

## Circuit

There is no circuit for this example, though your board must be connected to your computer via USB and the serial monitor window of the Arduino Software (IDE) should be open.



## Code

Here's a working example of several different concatenation examples :

```
/*
  Adding Strings together

 Examples of how to add strings together
 You can also add several different data types to string, as shown here:

 created 27 July 2010
 modified 2 Apr 2012
 by Tom Igoe

 http://www.arduino.cc/en/Tutorial/StringAdditionOperator

 This example code is in the public domain.
 */

// declare three strings:
String stringOne, stringTwo, stringThree;

void setup() {
 // initialize serial and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
 }

  stringOne = String("You added ");
  stringTwo = String("this string");
  stringThree = String();
 // send an intro:
 Serial.println("\n\nAdding strings together (concatenation):");
 Serial.println();
}

void loop() {
 // adding a constant integer to a string:
  stringThree =  stringOne + 123;
 Serial.println(stringThree);     // prints "You added 123"

 // adding a constant long interger to a string:
  stringThree = stringOne + 123456789;
 Serial.println(stringThree);     // prints "You added 123456789"
```

```
 // adding a constant character to a string:
  stringThree =  stringOne + 'A';
 Serial.println(stringThree);     // prints "You added A"

 // adding a constant string to a string:
  stringThree =  stringOne +  "abc";
 Serial.println(stringThree);     // prints "You added abc"

  stringThree = stringOne + stringTwo;
 Serial.println(stringThree);     // prints "You added this string"

 // adding a variable integer to a string:
 int sensorValue = analogRead(A0);
  stringOne = "Sensor value: ";
  stringThree = stringOne    + sensorValue;
 Serial.println(stringThree);     // prints "Sensor Value: 401" or whatever value analogRead(A0)
has

 // adding a variable long integer to a string:
 long currentTime = millis();
  stringOne = "millis() value: ";
  stringThree = stringOne + millis();
 Serial.println(stringThree);     // prints "The millis: 345345" or whatever value currentTime has

 // do nothing while true:
 while (true);
}
```

## StringIndexOf

String indexOf() and lastIndexOf() Method

The String object indexOf() method gives you the ability to search for the first instance of a particular character value in a String. You can also look for the first instance of the character after a given offset. The lastIndexOf() method lets you do the same things from the end of a String.

```
  String stringOne = "<HTML><HEAD><BODY>";

  int firstClosingBracket = stringOne.indexOf('>');
```

In this case, `firstClosingBracket` equals 5, because the first `>` character is at position 5 in the String (counting the first character as 0). If you want to get the second closing bracket, you can use the

fact that you know the position of the first one, and search from `firstClosingBracket + 1` as the offset, like so:

```
stringOne = "<HTML><HEAD><BODY>";

int secondClosingBracket = stringOne.indexOf('>', firstClosingBracket + 1 );
```

The result would be 11, the position of the closing bracket for the HEAD tag.

If you want to search from the end of the String, you can use the `lastIndexOf()` method instead. This function returns the position of the last occurrence of a given character.

```
stringOne = "<HTML><HEAD><BODY>";

int lastOpeningBracket = stringOne.lastIndexOf('<');
```

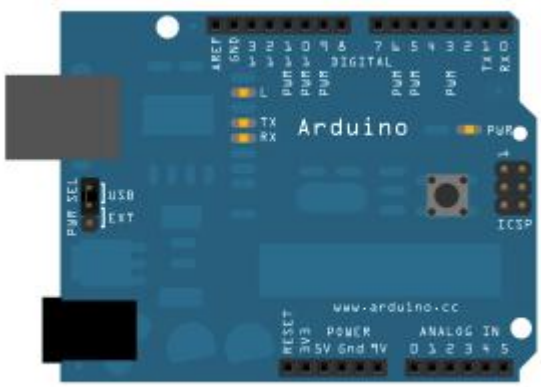
In this case, `lastOpeningBracket` equals 12, the position of the `<` for the BODY tag. If you want the opening bracket for the HEAD tag, it would be at `stringOne.lastIndexOf('<', lastOpeningBracket -1)`, or 6.

## Hardware Required

Arduino or Genuino Board

## Circuit

There is no circuit for this example, though your board must be connected to your computer via USB and the serial monitor window of the Arduino Software (IDE) should be open.




## Code

```
/*
  String indexOf() and lastIndexOf() functions

 Examples of how to evaluate, look for, and replace characters in a String
```

```
 created 27 July 2010
 modified 2 Apr 2012
 by Tom Igoe

 http://www.arduino.cc/en/Tutorial/StringIndexOf

 This example code is in the public domain.
 */

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
 }

 // send an intro:
 Serial.println("\n\nString indexOf() and lastIndexOf()   functions:");
 Serial.println();
}

void loop() {
 // indexOf() returns the position (i.e. index) of a particular character
 // in a string. For example, if you were parsing HTML tags, you could use it:
  String stringOne = "<HTML><HEAD><BODY>";
 int firstClosingBracket = stringOne.indexOf('>');
 Serial.println("The index of > in the string " + stringOne + " is " + firstClosingBracket);

  stringOne = "<HTML><HEAD><BODY>";
 int secondOpeningBracket = firstClosingBracket + 1;
 int secondClosingBracket = stringOne.indexOf('>', secondOpeningBracket);
 Serial.println("The index of   the second > in the string " + stringOne + " is
" + secondClosingBracket);

 // you can also use indexOf() to search for Strings:
  stringOne = "<HTML><HEAD><BODY>";
 int bodyTag = stringOne.indexOf("<BODY>");
 Serial.println("The index of the body tag in the string " + stringOne + " is " + bodyTag);

  stringOne = "<UL><LI>item<LI>item<LI>item</UL>";
 int firstListItem = stringOne.indexOf("<LI>");
 int secondListItem = stringOne.indexOf("<LI>", firstListItem + 1);
 Serial.println("The index of the second list tag in the string " + stringOne + " is
```

```
" + secondListItem);

 // lastIndexOf() gives you the last occurrence of a character or string:
 int lastOpeningBracket = stringOne.lastIndexOf('<');
 Serial.println("The index of the last < in the string " + stringOne + " is " + lastOpeningBracket);

 int lastListItem   = stringOne.lastIndexOf("<LI>");
 Serial.println("The index of the last list tag in the string " + stringOne + " is " + lastListItem);

 // lastIndexOf() can also search for a string:
  stringOne = "<p>Lorem ipsum dolor sit amet</p><p>Ipsem</p><p>Quod</p>";
 int lastParagraph = stringOne.lastIndexOf("<p");
 int secondLastGraf = stringOne.lastIndexOf("<p", lastParagraph - 1);
 Serial.println("The index of the second to last paragraph tag " + stringOne + " is
" + secondLastGraf);

 // do nothing while true:
 while (true);
}
```

## StringAppendOperator

Just as you can concatenate Strings with other data objects using the StringAdditionOperator, you can also use the +=operator and the concat() method to append things to Strings.
The += operator and the concat() method work the same way, it's just a matter of which style you prefer. The two examples below illustrate both, and result in the same String:

```
  String stringOne = "A long integer: ";

  // using += to add a long variable to a string:

  stringOne += 123456789;
```

or

```
  String stringTwo = "A long integer: ";

  // using concat() to add a long variable to a string:

  stringTwo.concat(123456789);
```

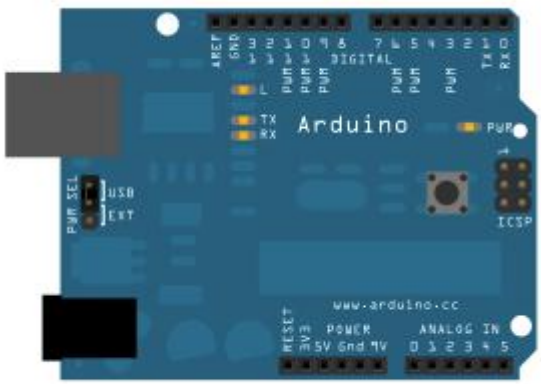
In both cases, stringOne equals "A long integer: 123456789". Like the + operator, these operators are handy for assembling longer strings from a combination of data objects.

## Hardware Required

Arduino or Genuino Board

## Circuit

There is no circuit for this example, though your board must be connected to your computer via USB and the serial monitor window of the Arduino Software (IDE) should be open.




## Code

```
/*
  Appending to Strings using the += operator and concat()

 Examples of how to append different data types to strings

 created 27 July 2010
 modified 2 Apr 2012
 by Tom Igoe

 http://www.arduino.cc/en/Tutorial/StringAppendOperator

 This example code is in the public domain.
 */

String stringOne, stringTwo;

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
   ; // wait for serial port to connect. Needed for native USB port only
 }

  stringOne = String("Sensor ");
  stringTwo = String("value");
```

```
 // send an intro:
 Serial.println("\n\nAppending to a string:");
 Serial.println();
}

void loop() {
 Serial.println(stringOne);  // prints    "Sensor "

 // adding a string to a string:
  stringOne += stringTwo;
 Serial.println(stringOne);  // prints "Sensor value"

 // adding a constant string to a string:
  stringOne += " for input ";
 Serial.println(stringOne);  // prints "Sensor value for input"

 // adding a constant character to a string:
  stringOne += 'A';
 Serial.println(stringOne);   // prints "Sensor value for input A"

 // adding a constant integer to a string:
  stringOne += 0;
 Serial.println(stringOne);   // prints "Sensor value for input A0"

 // adding a constant string to a string:
  stringOne += ": ";
 Serial.println(stringOne);  // prints "Sensor value for input"

 // adding a variable integer to a string:
  stringOne += analogRead(A0);
 Serial.println(stringOne);   // prints "Sensor value for input A0: 456" or whatever analogRead(A0)
is

 Serial.println("\n\nchanging the Strings' values");
  stringOne = "A long integer: ";
  stringTwo = "The millis(): ";

 // adding a constant long integer to a string:
  stringOne += 123456789;
 Serial.println(stringOne);   // prints "A long integer: 123456789"

 // using concat() to add a long variable to a string:
  stringTwo.concat(millis());
 Serial.println(stringTwo); // prints "The millis(): 43534" or whatever the value of the millis() is
```

```
 // do nothing while true:
 while (true);
}
```

## StringLengthTrim

String length() and trim() Commands

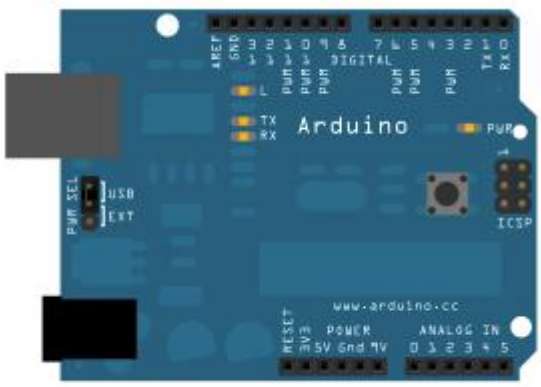
You can get the length of a Strings using the length() command, or eliminate extra characters using the trim() command. This example shows you how to use both commands.

### Hardware Required

Arduino or Genuino Board

### Circuit

There is no circuit for this example, though your board must be connected to your computer via USB and the serial monitor window of the Arduino Software (IDE) should be open.




### Code

trim() is useful for when you know there are extraneous whitespace characters on the beginning or the end of a String and you want to get rid of them. Whitespace refers to characters that take space but aren't seen. It includes the single space (ASCII 32), tab (ASCII 9), vertical tab (ASCII 11), form feed (ASCII 12), carriage return (ASCII 13), or newline (ASCII 10). The example below shows a String with whitespace, before and after trimming:

```
/*
  String length() and trim()

 Examples of how to use length() and trim() in a String

 created 27 July 2010
 modified 2 Apr 2012
```

```
 by Tom Igoe

 http://www.arduino.cc/en/Tutorial/StringLengthTrim

 This example code is in the public domain.
 */

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
 }

 // send an intro:
 Serial.println("\n\nString   length() and trim():");
 Serial.println();
}

void loop() {
 // here's a String with empty spaces at the end (called white space):
  String stringOne = "Hello!           ";
 Serial.print(stringOne);
 Serial.print("<--- end of string. Length: ");
 Serial.println(stringOne.length());

 // trim the white space off the string:
  stringOne.trim();
 Serial.print(stringOne);
 Serial.print("<--- end of trimmed string. Length: ");
 Serial.println(stringOne.length());

 // do nothing while true:
 while (true);
}
```

## StringCaseChanges

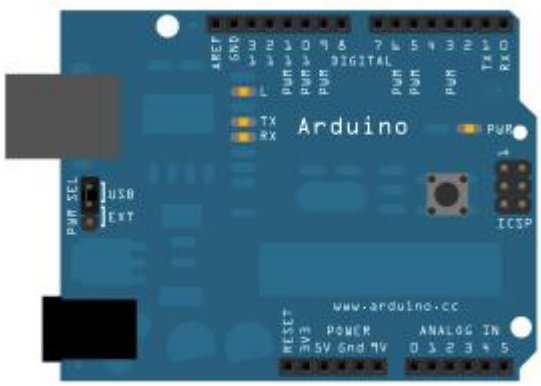
The String case change functions allow you to change the case of a String. They work just as their names imply. toUpperCase() changes the whole string to upper case characters, and toLowerCase() changes the whole String to lower case characters. Only the characters A to Z or a to z are affected.

Hardware Required

Arduino or Genuino Board

## Circuit

There is no circuit for this example, though your board must be connected to your computer via USB and the serial monitor window of the Arduino Software (IDE) should be open.




## Code

```
/*
  String Case changes

 Examples of how to change the case of a string

 created 27 July 2010
 modified 2 Apr 2012
 by Tom Igoe

 http://www.arduino.cc/en/Tutorial/StringCaseChanges

 This example code is in the public domain.
 */

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
 }

 // send an intro:
 Serial.println("\n\nString   case changes:");
 Serial.println();
}
```

```
void loop() {
 // toUpperCase() changes all letters to upper case:
  String stringOne = "<html><head><body>";
 Serial.println(stringOne);
  stringOne.toUpperCase();
 Serial.println(stringOne);

 // toLowerCase() changes all letters to lower case:
  String stringTwo = "</BODY></HTML>";
 Serial.println(stringTwo);
  stringTwo.toLowerCase();
 Serial.println(stringTwo);


 // do nothing while true:
 while (true);
}
```

## StringReplace

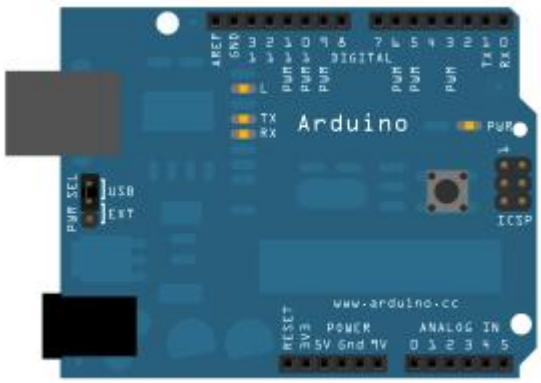
The String replace() function allows you to replace all instances of a given character with another character. You can also use replace to replace substrings of a string with a different substring.

### Hardware Required

Arduino or Genuino Board

### Circuit

There is no circuit for this example, though your board must be connected to your computer via USB and the serial monitor window of the Arduino Software (IDE) should be open.




### Code

Caution: If you try to replace a substring that's more than the whole string itself, nothing will be replaced. For example:

```
  String stringOne = "<html><head><body>";

  String stringTwo = stringOne.replace("<html><head></head><body></body></html>",
"Blah");
```

In this case, the code will compile, but stringOne will remain unchanged, since the replacement substring is more than the String itself.

```
/*
  String replace()

 Examples of how to replace characters or substrings of a string

 created 27 July 2010
 modified 2 Apr 2012
 by Tom Igoe

 http://www.arduino.cc/en/Tutorial/StringReplace

 This example code is in the public domain.
 */

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
   ; // wait for serial port to connect. Needed for native USB port only
 }

 // send an intro:
 Serial.println("\n\nString   replace:\n");
 Serial.println();
}

void loop() {
  String stringOne = "<html><head><body>";
 Serial.println(stringOne);
 // replace() changes all instances of one substring with another:
 // first, make a copy of th original string:
  String stringTwo = stringOne;
 // then perform the replacements:
  stringTwo.replace("<", "</");
 // print the original:
 Serial.println("Original string: " + stringOne);
```

```
 // and print the modified string:
 Serial.println("Modified string: " + stringTwo);

 // you can also use replace() on single characters:
  String normalString = "bookkeeper";
 Serial.println("normal: " + normalString);
  String leetString = normalString;
  leetString.replace('o', '0');
  leetString.replace('e', '3');
 Serial.println("l33tspeak: " + leetString);

 // do nothing while true:
 while (true);
}
```

## StringRemove

The remove() method of the String class allows you to remove a specific part of a String. It can be used with one or two arguments. With one argument, the string from that index to the end is removed. With two arguments, the first argument is the index of the start of the cut, and the second argument is the length of the cut.

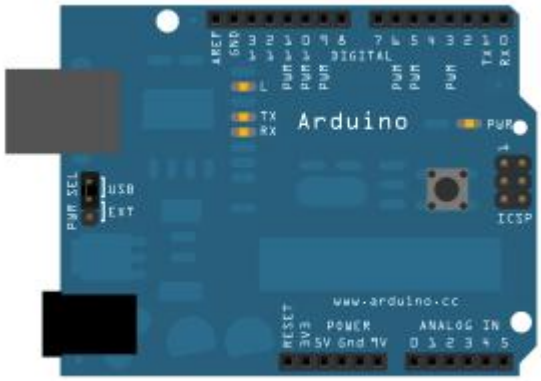
In this example, the Arduino prints on the serial monitor a full string and the same string with a portion removed. Both ways of calling the method are demonstrated.

Hardware Required:

Arduino Board

Circuit

There is no circuit for this example, though your Arduino must be connected to your computer via USB.



Code

```
/*
  Example of the String remove() method

 Print on the serial monitor a full string, and then the string with a portion removed.
 Both removal methods are demonstrated.

 The circuit:
 No external components needed.

 created 10 Nov 2014
 by Arturo Guadalupi

 This example code is in the public domain.
 */

String exString = "Hello World!";     // example string

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
    ; // wait for serial port to connect. Needed for Leonardo only
 }

 // send an intro:
 Serial.println("\n\nString remove() method example");
 Serial.println();
}

void loop() {
 // Print the initial string
 Serial.println("The full string:");
 Serial.println(exString);

 // Removing from an index through the end
  exString.remove(7);   // Remove from from index=7 through the end of the string
 Serial.println("String after removing from the seventh index through the end");
 Serial.println(exString);  // Should be just "Hello W"

 // Removing only a portion in the middle of a string
  exString = "Hello World!";
  exString.remove(2, 6); // Remove six characters starting at index=2
 Serial.println("String after removing six characters starting at the third position");
```

```
  Serial.println(exString); // Should be just "Herld!"

  Serial.println();
  Serial.println();

  while(1)
    ; // no need to do it again
}
```

## StringCharacters

The String functions charAt() and setCharAt() are used to get or set the value of a character at a given position in a String.

At their simplest, these functions help you search and replace a given character. For example, the following replaces the colon in a given String with an equals sign:

```
String reportString = "SensorReading: 456";

int colonPosition = reportString.indexOf(':');

reportString.setCharAt(colonPosition, '=');
```

Here's an example that checks to see if the first letter of the second word is 'B':

```
String reportString = "Franklin, Benjamin";

int spacePosition = reportString.indexOf(' ');

if (reportString.charAt(spacePosition + 1) == 'B') {

      Serial.println("You might have found the Benjamins.")

}
```

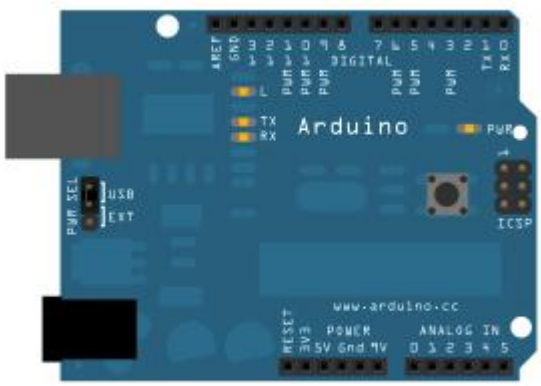
Caution: If you try to get the charAt or try to setCharAt() a value that's longer than the String's length, you'll get unexpected results. If you're not sure, check to see that the position you want to set or get is less than the string's length using the length() function.

Hardware Required

Arduino or Genuino Board

## Circuit

There is no circuit for this example, though your board must be connected to your computer via USB and the serial monitor window of the Arduino Software (IDE) should be open.




## Code

```
/*
  String charAt() and setCharAt()

 Examples of how to get and set characters of a String

 created 27 July 2010
 modified 2 Apr 2012
 by Tom Igoe

 http://www.arduino.cc/en/Tutorial/StringCharacters

 This example code is in the public domain.
 */

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
 }

 Serial.println("\n\nString   charAt() and setCharAt():");
}

void loop() {
 // make a string to report a sensor reading:
```

```
 String reportString = "SensorReading: 456";
Serial.println(reportString);

// the reading's most significant digit is at position 15 in the reportString:
char mostSignificantDigit = reportString.charAt(15);

 String message = "Most significant digit of the sensor reading is: ";
Serial.println(message + mostSignificantDigit);

// add blank space:
Serial.println();

// you can alo set the character of a string. Change the : to a = character
 reportString.setCharAt(13, '=');
Serial.println(reportString);

// do nothing while true:
while (true);
}
```

## StringStartsWithEndsWith

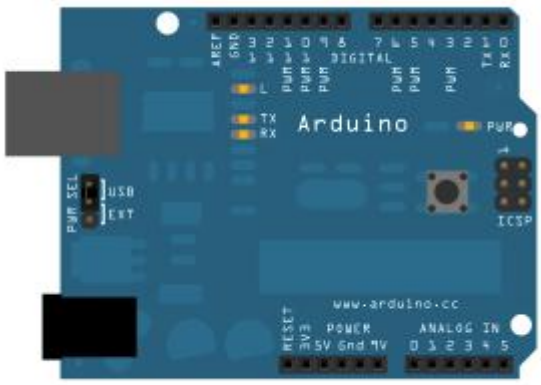
The String functions startsWith() and endsWith() allow you to check what character or substring a given String starts or ends with. They're basically special cases of substring.

### Hardware Required

Arduino or Genuino Board

### Circuit

There is no circuit for this example, though your board must be connected to your computer via USB and the serial monitor window of the Arduino Software (IDE) should be open..




### Code

startsWith() and endsWith() can be used to look for a particular message header, or for a single character at the end of a String. They can also be used with an offset to look for a substring starting at a particular position. For example:

```
stringOne = "HTTP/1.1 200 OK";

  if (stringOne.startsWith("200 OK", 9)) {

     Serial.println("Got an OK from the server");

  }
```

This is functionally the same as this:

```
stringOne = "HTTP/1.1 200 OK";

  if (stringOne.substring(9) == "200 OK") {

     Serial.println("Got an OK from the server");

  }
```

Caution: If you look for a position that's outside the range of the string,you'll get unpredictable results. For example, in the example above stringOne.startsWith("200 OK", 16) wouldn't check against the String itself, but whatever is in memory just beyond it. For best results, make sure the index values you use for startsWith and endsWith are between 0 and the String's length().

```
/*
  String startWith() and endsWith()

 Examples of how to use startsWith() and endsWith() in a String

 created 27 July 2010
 modified 2 Apr 2012
 by Tom Igoe

 http://www.arduino.cc/en/Tutorial/StringStartsWithEndsWith

 This example code is in the public domain.
 */

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
```

```
 while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
 }

 // send an intro:
 Serial.println("\n\nString startsWith() and endsWith():");
 Serial.println();
}

void loop() {
 // startsWith() checks to see if a String starts with a particular substring:
  String stringOne = "HTTP/1.1 200 OK";
 Serial.println(stringOne);
 if (stringOne.startsWith("HTTP/1.1")) {
    Serial.println("Server's using http version 1.1");
 }

 // you can also look for startsWith() at an offset position in the string:
  stringOne = "HTTP/1.1 200 OK";
 if (stringOne.startsWith("200 OK", 9)) {
    Serial.println("Got an OK from the server");
 }

 // endsWith() checks to see if a String ends with a particular character:
  String sensorReading = "sensor = ";
  sensorReading += analogRead(A0);
 Serial.print(sensorReading);
 if (sensorReading.endsWith("0")) {
    Serial.println(". This reading is divisible by ten");
 } else {
    Serial.println(". This reading is not divisible by ten");
 }

 // do nothing while true:
 while (true);
}
```

## StringComparisonOperators

The String comparison operators ==, !=,>, < ,>=, <= , and the equals() and equalsIgnoreCase()methods allow you to make alphabetic comparisons between Strings. They're useful for sorting and alphabetizing, among other things.

The operator == and the method equals() perform identically. In other words,

```
if (stringOne.equals(stringTwo)) {
```

is identical to

```
if (stringOne ==stringTwo) {
```

The ">" (greater than) and "<" (less than) operators evaluate strings in alphabetical order, on the first character where the two differ. So, for example `"a" < "b"` and `"1" < "2"`, but `"999" > "1000"` because 9 comes after 1.

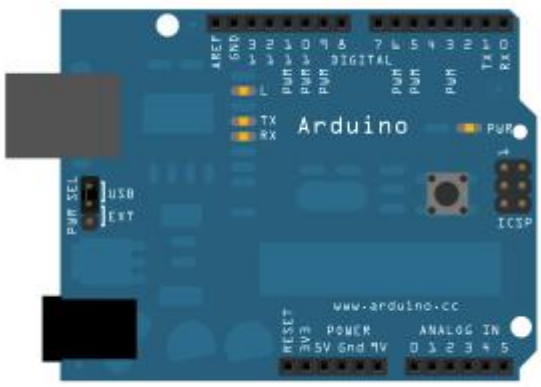
Caution: String comparison operators can be confusing when you're comparing numeric strings, because the numbers are treated as strings and not as numbers. If you need to compare numbers, compare them as ints, floats, or longs, and not as Strings.

## Hardware Required

Arduino or Genuino Board

## Circuit

There is no circuit for this example, though your board must be connected to your computer via USB and the serial monitor window of the Arduino Software (IDE) should be open.




## Code

```
/*
  Comparing Strings

 Examples of how to compare strings using the comparison operators

 created 27 July 2010
 modified 2 Apr 2012
 by Tom Igoe

 http://www.arduino.cc/en/Tutorial/StringComparisonOperators
```

```
 This example code is in the public domain.
 */

String stringOne, stringTwo;

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
 }


  stringOne = String("this");
  stringTwo = String("that");
 // send an intro:
 Serial.println("\n\nComparing Strings:");
 Serial.println();

}

void loop() {
 // two strings equal:
 if (stringOne == "this") {
    Serial.println("StringOne == \"this\"");
 }
 // two strings not equal:
 if (stringOne != stringTwo) {
    Serial.println(stringOne + " =! " + stringTwo);
 }

 // two strings not equal (case sensitivity matters):
  stringOne = "This";
  stringTwo = "this";
 if (stringOne != stringTwo) {
    Serial.println(stringOne + " =! " + stringTwo);
 }
 // you can also use equals() to see if two strings are the same:
 if (stringOne.equals(stringTwo)) {
    Serial.println(stringOne + " equals " + stringTwo);
 } else {
    Serial.println(stringOne + " does not equal " + stringTwo);
 }
```

```
// or perhaps you want to ignore case:
if (stringOne.equalsIgnoreCase(stringTwo)) {
  Serial.println(stringOne + " equals (ignoring case) " + stringTwo);
} else {
  Serial.println(stringOne + " does not equal (ignoring case) " + stringTwo);
}

// a numeric string compared to the number it represents:
 stringOne = "1";
int numberOne = 1;
if (stringOne.toInt() == numberOne) {
  Serial.println(stringOne + " = " + numberOne);
}


// two numeric strings compared:
 stringOne = "2";
 stringTwo = "1";
if (stringOne >= stringTwo) {
  Serial.println(stringOne + " >= " + stringTwo);
}

// comparison operators can be used to compare strings for alphabetic sorting too:
 stringOne = String("Brown");
if (stringOne < "Charles") {
  Serial.println(stringOne + " < Charles");
}

if (stringOne > "Adams") {
  Serial.println(stringOne + " > Adams");
}

if (stringOne <= "Browne") {
  Serial.println(stringOne + " <= Browne");
}


if (stringOne >= "Brow") {
  Serial.println(stringOne + " >= Brow");
}

// the compareTo() operator also allows you to compare strings
// it evaluates on the first character that's different.
// if the first character of the string you're comparing to
```

```
// comes first in alphanumeric order, then compareTo() is greater than 0:
 stringOne = "Cucumber";
 stringTwo = "Cucuracha";
if (stringOne.compareTo(stringTwo) < 0) {
   Serial.println(stringOne + " comes before " + stringTwo);
} else {
   Serial.println(stringOne + " comes after " + stringTwo);
}

delay(10000);  // because the next part is a loop:

// compareTo() is handy when you've got strings with numbers in them too:

while (true) {
    stringOne = "Sensor: ";
    stringTwo = "Sensor: ";

    stringOne += analogRead(A0);
    stringTwo += analogRead(A5);

   if (stringOne.compareTo(stringTwo) < 0) {
     Serial.println(stringOne + " comes before " + stringTwo);
   } else {
     Serial.println(stringOne + " comes after " + stringTwo);

   }
 }
}
```

## StringSubstring

The String function substring() is closely related to charAt(), startsWith() and endsWith(). It allows you to look for an instance of a particular substring within a given String.

### Hardware Required

Arduino or Genuino Board

### Circuit

There is no circuit for this example, though your board must be connected to your computer via USB and the serial monitor window of the Arduino Software (IDE) should be open.


## Code

substring() with only one parameter looks for a given substring from the position given to the end of the string. It expects that the substring extends all the way to the end of the String. For example:

```
String stringOne = "Content-Type: text/html";



// substring(index) looks for the substring from the index position to the end:

if (stringOne.substring(19) == "html") {

  }
```

is true, while

```
String stringOne = "Content-Type: text/html";



// substring(index) looks for the substring from the index position to the end:

if (stringOne.substring(19) == "htm") {

  }
```

is not true, because there's an l after the htm in the String.

substring() with two parameters looks for a given substring from the first parameter to the second. For example:

```
  String stringOne = "Content-Type: text/html";

  // you can also look for a substring in the middle of a string:

  if (stringOne.substring(14,18) == "text") {

  }
```

This looks for the word `text` from positions 14 through 18 of the String.

Caution: make sure your index values are within the String's length or you'll get unpredictable results. This kind of error can be particularly hard to find with the second instance of `substring()` if the starting position is less than the String's length, but the ending position isn't.

```
/*
  String substring()

 Examples of how to use substring in a String

 created 27 July 2010,
 modified 2 Apr 2012
 by Zach Eveland

 http://www.arduino.cc/en/Tutorial/StringSubstring

 This example code is in the public domain.
 */

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
   ; // wait for serial port to connect. Needed for native USB port only
 }

 // send an intro:
 Serial.println("\n\nString   substring():");
 Serial.println();
}

void loop() {
 // Set up a String:
  String stringOne = "Content-Type: text/html";
 Serial.println(stringOne);
```

```
 // substring(index) looks for the substring from the index position to the end:
 if (stringOne.substring(19) == "html") {
   Serial.println("It's an html file");
 }
 // you can also look for a substring in the middle of a string:
 if (stringOne.substring(14, 18) == "text") {
   Serial.println("It's a text-based file");
 }

 // do nothing while true:
 while (true);
}
```

## 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()

### Description

Converts a value to the char data type.

### Syntax

char(x)

### Parameters

x: a value of any type

### Returns

Char

## byte()

### Description

Converts a value to the byte data type.

### Syntax

byte(x)

### Parameters

x: a value of any type

### Returns

byte

## int()

### Description

Converts a value to the int data type.

### Syntax

int(x)

### Parameters

x: a value of any type

### Returns

int

## word()

### Description

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()

### Description

Converts a value to the long data type.

### Syntax

long(x)

### Parameters

x: a value of any type

### Returns

long

## float()

### Description

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 early versions of languages such as BASIC where every variable is a global variable.

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

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

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

### Example:

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

```
*/

#define randomWalkLowRange -20

#define randomWalkHighRange 20

int stepsize;

int thisTime;

int total;

void setup()

{

   Serial.begin(9600);

}

void loop()

{              //    tetst randomWalk function

   stepsize = 5;

   thisTime = randomWalk(stepsize);

   Serial.println(thisTime);

     delay(10);

}

int randomWalk(int moveSize){

   static int    place;          // variable to store value in random walk - declared static so that it
stores

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

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

```
  if (place < randomWalkLowRange){                              // check lower and upper limits

    place = 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

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

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

### Description

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

### Syntax

sizeof(variable)

### Parameters

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

### Example code

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

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

```
char myStr[] = "this is a test";
int i;
void setup(){
 Serial.begin(9600);
}

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

```
 delay(5000); // slow down the program
}
```

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

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

## PROGMEM

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

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

PROGMEM is part of the pgmspace.h library that is available in the AVR architecture only. So you first need to include the library at the top your sketch, like this:

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

### Syntax

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

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

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

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

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

Using PROGMEM is also a two-step procedure. After getting the data into Flash memory, it requires special methods (functions), also defined in the pgmspace.h library, to read the data from program memory back into SRAM, so we can do something useful with it.

### Example

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

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


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

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

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


void setup() {
 Serial.begin(9600);
 while (!Serial);

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

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

```
 Serial.println();
}

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

}
```

Arrays of strings

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

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

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

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

 Setting up a table (array) of strings in program memory is slightly complicated, but
 here is a good template to follow.

 Setting up the strings is a two-step process. First define the strings.
*/

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

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

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

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

void setup()
{
 Serial.begin(9600);
 while(!Serial);
 Serial.println("OK");
}

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


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

Note

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

The following code will NOT work when inside a function:

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

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

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

### The F() macro

When an instruction like :

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

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

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

# Functions

## Digital I/O

### pinMode()

#### Description

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

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

#### Syntax

pinMode(pin, mode)

#### Parameters

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

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

#### Returns

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()

Description

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

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

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

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

Syntax

digitalWrite(pin, value)

Parameters

pin: the pin number

value: HIGH or LOW

Returns

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

}
```

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

### Note

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

## digitalRead()

### Description

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

### Syntax

digitalRead(pin)

### Parameters

pin: the number of the digital pin you want to read (int)

### Returns

HIGH or LOW

### Example

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

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

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

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

### Note

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

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

# Analog I/O

## analogReference()

### Description

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 (not available on the Arduino Mega)
- INTERNAL1V1: a built-in 1.1V reference (Arduino Mega only)
- INTERNAL2V56: a built-in 2.56V reference (Arduino Mega only)

- EXTERNAL: the voltage applied to the AREF pin (0 to 5V only) is used as the reference.

### Syntax

analogReference(type)

### Parameters

type: which type of reference to use (DEFAULT, INTERNAL, INTERNAL1V1, INTERNAL2V56, or EXTERNAL).

### Returns

None.

### Note

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

### Warning

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

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

## analogRead()

### Description

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

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

### Syntax

analogRead(pin)

### Parameters

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

### Returns

int (0 to 1023)

### 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.).

### Example

```
int analogPin = 3;       // potentiometer wiper (middle terminal) connected to analog pin 3

                         // outside leads to ground and +5V
int val = 0;             // variable to store the value read

void setup()
{

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

void loop()
{

  val = analogRead(analogPin);     // read the input pin

  Serial.println(val);               // debug value
}
```

## analogWrite()

### Description

Writes an analog value (PWM wave) to a pin. Can be used to light a LED at varying brightnesses or drive a motor at various speeds. After a call to analogWrite(), the pin will generate a steady square wave of the specified duty cycle until the next call to analogWrite() (or a call

to digitalRead() or digitalWrite() on the same pin). The frequency of the PWM signal on most pins is approximately 490 Hz. On the Uno and similar boards, pins 5 and 6 have a frequency of approximately 980 Hz. Pins 3 and 11 on the Leonardo also run at 980 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 - 13 and 44 - 46. Older Arduino boards with an ATmega8 only support analogWrite() on pins 9, 10, and 11.

The Arduino Due supports analogWrite() on pins 2 through 13, plus pins DAC0 and DAC1. Unlike the PWM pins, DAC0 and DAC1 are Digital to Analog converters, and act as true analog outputs.

You do not need to call pinMode() to set the pin as an output before calling analogWrite().

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

### Syntax

analogWrite(pin, value)

### Parameters

pin: the pin to write to.

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
}
```

# Due & Zero only

## analogReadResolution()

### Description

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

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

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

### Syntax

analogReadResolution(bits)

### Parameters

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

### Returns

None.

### Note

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

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

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

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

Example

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

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

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

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

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

 // a little delay to not hog serial monitor
 delay(100);
}
```

# analogWriteResolution()

## Description

analogWriteResolution() is an extension of the Analog API for the Arduino Due, Genuino and Arduino Zero and MKR1000.

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

The Due has the following hardare capabilities:

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

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

The Zero has the following hardare capabilities:

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

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

The MKR1000 has the following hardare capabilities:

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

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

## Syntax

analogWriteResolution(bits)

## Parameters

bits: determines the resolution (in bits) of the values used in the analogWrite() function. The value can range from 1 to 32. If you choose a resolution higher or lower than your board's

hardware capabilities, the value used in analogWrite() will be either truncated if it's too high or padded with zeros if it's too low. See the note below for details.

### Returns

None.

### Note

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

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

### Example

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

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

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

 // change the PWM resolution to 12 bits
 // the full 12 bit resolution is only supported
 // on the Due
```

```
 analogWriteResolution(12);
 analogWrite(12, map(sensorVal, 0, 1023, 0, 4095));
 Serial.print(" , 12-bit PWM value : ");
 Serial.print(map(sensorVal, 0, 1023, 0, 4095));

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

 delay(5);
}
```

# Advanced I/O

## tone()

### Description

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

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

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

| Board | Min frequency (Hz) | Max frequency (Hz) |
|---|---|---|
| Uno, Mega, Leonardo and other AVR boards | 31 | 65535 |
| Gemma | Not implemented | Not implemented |
| Zero | 41 | 275000 |
| Due | Not implemented | Not implemented |

For technical details, see Brett Hagman's notes.

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

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

### Parameters

pin: the pin on which to generate the tone

frequency: the frequency of the tone in hertz - unsigned int

duration: the duration of the tone in milliseconds (optional) - unsigned long

### Returns

nothing

## noTone()

### Description

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()

### Description

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

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

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

### Syntax

shiftOut(dataPin, clockPin, bitOrder, value)

### Parameters

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

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

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

value: the data to shift out. (byte)

### Returns

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(dataPin, clock, MSBFIRST, data);

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

```
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

## 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);
 }
}
```

## shiftIn()

### Description

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

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

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

### Syntax

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

### Parameters

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

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

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

### Returns

the value read (byte)

## pulseIn()

### Description

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

The timing of this function has been determined empirically and will probably show errors in shorter pulses. Works on pulses from 10 microseconds to 3 minutes in length. Please also note that if the pin is already high when the function is called, it will wait for the pin to go LOW and

then HIGH before it starts counting. This routine can be used only if interrupts are activated. Furthermore the highest resolution is obtained with short intervals.

### Syntax

pulseIn(pin, value)
pulseIn(pin, value, timeout)

### Parameters

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

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

timeout (optional): the number of microseconds to wait for the pulse to be completed: the function returns 0 if no complete pulse was received within the timeout. Default is one second (unsigned long).

### Returns

the length of the pulse (in microseconds) or 0 if no pulse is completed 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()

### Description

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

### Parameters

None

### Returns

Number of milliseconds since the program started (*unsigned long*)

### Note:

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

### Example

```
unsigned long time;

void setup(){
 Serial.begin(9600);
}
void loop(){
 Serial.print("Time: ");
  time = millis();
 //prints time since program started
 Serial.println(time);
 // wait a second so as not to send massive amounts of data
 delay(1000);
}
```

## micros()

### Description

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

Note: there are 1,000 microseconds in a millisecond and 1,000,000 microseconds in a second.

### Parameters

None

### Returns

Number of microseconds since the program started (unsigned long)

### Example

```
unsigned long time;

void setup(){

  Serial.begin(9600);

}

void loop(){

  Serial.print("Time: ");

  time = micros();

  //prints time since program started

  Serial.println(time);

  // wait a second so as not to send massive amounts of data

  delay(1000);

}
```

## delay()

### Description

Pauses the program for the amount of time (in miliseconds) 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()

### Description

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

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

### Syntax

delayMicroseconds(us)

### Parameters

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

### Returns

None

### Example

```
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
}
```

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

### 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)

### Description

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

min(a, 100);
a++;             // 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(senVal, 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

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

## abs(x)

### Description

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

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

## constrain(x, a, b)

### Description

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)

### Description

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

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

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

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

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

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

is also valid and works well.

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

### 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);
}
```

### Appendix

For the mathematically inclined, here's the whole function

```
long map(long x, long in_min, long in_max, long out_min, long out_max)

{

   return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;

}
```

## pow(base, exponent)

### Description

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

### 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)

### Description

Calculates the square root of a number.

### Parameters

x: the number, any data type

### Returns

double, the number's square root.

# Trigonometry

## sin(rad)

### Description

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)

Description

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)

Description

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)

# Characters

## isAlphaNumeric(thisChar)

Description

Analyse if a char is alphanumeric.

Parameters

thisChar: the char to be analysed.

Returns

True or false.

## isAlpha(thisChar)

Description

Analyse if a char is is alpha.

Parameters

thisChar: the char to be analysed

Returns

True or false.

## isAscii(thisChar)

Description

Analyse if a char is ASCII.

Parameters

thisChar: the char to be analysed

Returns

True or false.

## isWhitespace(thisChar)

Description

Analyse if a char is a white space.

Parameters

thisChar: the char to be analysed

Returns

True or false.

## isControl(thisChar)

Description

Analyse if a char is a control character.

Parameters

thisChar: the char to be analysed

Returns

True or false.

## isDigit(thisChar)

Description

Analyse if a char is a digit.

Parameters

thisChar: the char to be analysed

Returns

True or false.

## isGraph(thisChar)

Description

Analyse if a char is a printable character.

Parameters

thisChar: the char to be analysed

Returns

True or false.

## isLowerCase(thisChar)

Description

Analyse if a char is a lower case character.

Parameters

thisChar: the char to be analysed

Returns

True or false.

## isPrintable(thisChar)

Description

Analyse if a char is a printable character.

Parameters

thisChar: the char to be analysed

Returns

True or false.

## isPunct(thisChar)

Description

Analyse if a char is punctuation character.

Parameters

thisChar: the char to be analysed

Returns

True or false.

## isSpace(thisChar)

Description

Analyse if a char is a space character.

Parameters

thisChar: the char to be analysed

Returns

True or false.

## isUpperCase(thisChar)

### Description

Analyse if a char is an upper case character.

### Parameters

thisChar: the char to be analysed

### Returns

True or false.

## isHexadecimalDigit(thisChar)

### Description

Analyse if a char is a valid hexadecimal digit.

### Parameters

thisChar: the char to be analysed

### Returns

True or false.

### Examples

```
/*
  Character analysis operators

 Examples using the character analysis operators.
 Send any byte and the sketch will tell you about it.

 created 29 Nov 2010
 modified 2 Apr 2012
 by Tom Igoe

 This example code is in the public domain.
 */

void setup() {
 // Open serial communications and wait for port to open:
 Serial.begin(9600);
 while (!Serial) {
```

```
    ; // wait for serial port to connect. Needed for native USB port only
  }

  // send an intro:
  Serial.println("send any byte and I'll tell you everything I can about it");
  Serial.println();
}

void loop() {
  // get any incoming bytes:
  if (Serial.available() > 0) {
    int thisChar = Serial.read();

    // say what was sent:
    Serial.print("You sent me: \'");
    Serial.write(thisChar);
    Serial.print("\'  ASCII Value: ");
    Serial.println(thisChar);

    // analyze what was sent:
    if (isAlphaNumeric(thisChar)) {
      Serial.println("it's alphanumeric");
    }
    if (isAlpha(thisChar)) {
      Serial.println("it's alphabetic");
    }
    if (isAscii(thisChar)) {
      Serial.println("it's ASCII");
    }
    if (isWhitespace(thisChar)) {
      Serial.println("it's whitespace");
    }
    if (isControl(thisChar)) {
      Serial.println("it's a control character");
    }
    if (isDigit(thisChar)) {
      Serial.println("it's a numeric digit");
    }
    if (isGraph(thisChar)) {
      Serial.println("it's a printable character that's not whitespace");
    }
    if (isLowerCase(thisChar)) {
      Serial.println("it's lower case");
    }
```

```
    if (isPrintable(thisChar)) {
      Serial.println("it's printable");
    }
    if (isPunct(thisChar)) {
      Serial.println("it's punctuation");
    }
    if (isSpace(thisChar)) {
      Serial.println("it's a space character");
    }
    if (isUpperCase(thisChar)) {
      Serial.println("it's upper case");
    }
    if (isHexadecimalDigit(thisChar)) {
      Serial.println("it's a valid hexadecimaldigit (i.e. 0 - 9, a - F, or A - F)");
    }

    // add some space and ask for another byte:
    Serial.println();
    Serial.println("Give me another byte:");
    Serial.println();
  }
}
```

# Random Numbers

## randomSeed(seed)

### Description

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

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

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

### Parameters

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()

### Description

The random function generates pseudo-random numbers.

### Syntax

random(max)
random(min, max)

### Parameters

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

### Description

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

### Syntax

lowByte(x)

### Parameters

x: a value of any type

### Returns

byte

## highByte()

### Description

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

### Syntax

highByte(x)

### Parameters

x: a value of any type

### Returns

byte

## bitRead()

### Description

Reads a bit of a number.

### Syntax

bitRead(x, n)

### Parameters

x: the number from which to read

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

### Returns

the value of the bit (0 or 1).

## bitWrite()

### Description

Writes a bit of a numeric variable.

### Syntax

bitWrite(x, n, b)

### Parameters

x: the numeric variable to which to write

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

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

### Returns

none

## bitSet()

### Description

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()

### Description

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

### Syntax

bitClear(x, n)

### Parameters

x: the numeric variable whose bit to clear

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

### Returns

none

## bit()

### Description

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

### Syntax

bit(n)

### Parameters

n: the bit whose value to compute

### Returns

the value of the bit

# External Interrupts

# attachInterrupt()

## Description

Digital Pins With Interrupts

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

| Board | Digital Pins Usable For Interrupts |
|---|---|
| Uno, Nano, Mini, other 328-based | 2, 3 |
| Mega, Mega2560, MegaADK | 2, 3, 18, 19, 20, 21 |
| Micro, Leonardo, other 32u4-based | 0, 1, 2, 3, 7 |
| Zero | all digital pins, except 4 |
| MKR1000 Rev.1 | 0, 1, 4, 5, 6, 7, 8, 9, A1, A2 |
| Due | all digital pins |
| 101 | all digital pins |

## Note

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

## Using Interrupts

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

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

## About Interrupt Service Routines

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

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

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

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

## Syntax

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

## Parameters

| | | |
|---|---|---|
| interrupt: | the number of the interrupt (int) | |
| pin: | the pin number | (Arduino Due, Zero, MKR1000 only) |
| ISR: | the ISR to call when the interrupt occurs; this function must take no parameters and return nothing. This function is sometimes referred to as an interrupt service routine. | |
| mode: | defines when the interrupt should be triggered. Four constants are predefined as valid values:<br>● LOW to trigger the interrupt whenever the pin is low,<br>● CHANGE to trigger the interrupt whenever the pin changes value<br>● RISING to trigger when the pin goes from low to high,<br>● FALLING for when the pin goes from high to low. | |

The Due board allows also:

- HIGH to trigger the interrupt whenever the pin is high. (Arduino Due, Zero, MKR1000 only)

## Returns

none

## Example

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

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

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

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

## Interrupt numbers

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

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

However on newer boards with high performances processors like:

- Due;
- Zero;

- MKR1000;
- 101;

a one-to-one mapping between the pin and the interrupt number exist, so the syntaxes:

- attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
- attachInterrupt(interruptPin, blink, CHANGE);

are equivalent.

| Board | int.0 | int.1 | int.2 | int.3 | int.4 | int.5 |
|---|---|---|---|---|---|---|
| Uno, Ethernet | 2 | 3 | | | | |
| Mega2560 | 2 | 3 | 21 | 20 | 19 | 18 |
| 32u4 based (e.g Leonardo, Micro) | 3 | 2 | 0 | 1 | 7 | |
| Due, Zero, MKR1000, 101 | interrupt number = pin number | | | | | |

## detachInterrupt()

### Description

Turns off the given interrupt.

### Syntax

detachInterrupt(interrupt)
detachInterrupt(digitalPinToInterrupt(pin));
detachInterrupt(pin) (Arduino Due, Zero only)

### Parameters

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

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

# Interrupts

## interrupts()

### Description

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

### Parameters

None

### Returns

None

### Example

```
void setup() {}

void loop()

{

  noInterrupts();

  // critical, time-sensitive code here

  interrupts();

  // other code here

}
```

## noInterrupts()

### Description

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

### Parameters

None.

Returns

None.

Example

```
void setup() {}

void loop()

{

   noInterrupts();

   // critical, time-sensitive code here

   interrupts();

   // other code here

}
```

# Communication

## Serial

Serial communication on pins TX/RX uses TTL logic levels (5V or 3.3V depending on the board). Don't connect these pins directly to an RS232 serial port; they operate at +/- 12V and can damage your Arduino board.

Serial is 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.

The Arduino Due has three additional 3.3V TTL serial ports: Serial1 on pins 19 (RX) and 18 (TX); Serial2 on pins 17 (RX) and 16 (TX), Serial3 on pins 15 (RX) and 14 (TX). Pins 0 and 1 are also connected to the corresponding pins of the ATmega16U2 USB-to-TTL Serial chip, which is connected to the USB debug port. Additionally, there is a native USB-serial port on the SAM3X chip, SerialUSB'.

The Arduino Leonardo board uses Serial1 to communicate via TTL (5V) serial on pins 0 (RX) and 1 (TX). Serial is reserved for USB CDC

## Stream

Stream is the base class for character and binary based streams. It is not called directly, but invoked whenever you use a function that relies on it.

Stream defines the reading functions in Arduino. When using any core functionality that uses a read() or similar method, you can safely assume it calls on the Stream class. For functions like print(), Stream inherits from the Print class.

Some of the libraries that rely on Stream include :

## Wire Library

This library allows you to communicate with I2C / TWI devices. On the Arduino boards with the R3 layout (1.0 pinout), the SDA (data line) and SCL (clock line) are on the pin headers close to the AREF pin. The Arduino Due has two I2C / TWI interfaces SDA1 and SCL1 are near to the AREF pin and the additional one is on pins 20 and 21.

As a reference the table below shows where TWI pins are located on various Arduino boards.

| Board | I2C / TWI pins |
|---|---|
| Uno, Ethernet | A4 (SDA), A5 (SCL) |
| Mega2560 | 20 (SDA), 21 (SCL) |
| Leonardo | 2 (SDA), 3 (SCL) |
| Due | 20 (SDA), 21 (SCL), SDA1, SCL1 |

As of Arduino 1.0, the library inherits from the Stream functions, making it consistent with other read/write libraries. Because of this, send() and receive() have been replaced with read() and write().

Note

There are both 7- and 8-bit versions of I2C addresses. 7 bits identify the device, and the eighth bit determines if it's being written to or read from. The Wire library uses 7 bit addresses throughout. If you have a datasheet or sample code that uses 8 bit address, you'll want to drop the low bit (i.e. shift the value one bit to the right), yielding an address between 0 and 127. However the addresses from 0 to 7 are not used because are reserved so the first address that can be used is 8. Please note that a pull-up resistor is needed when connecting SDA/SCL pins. Please refer to the examples for more informations. MEGA 2560 board has pull-up resistors on pins 20 - 21 onboard.

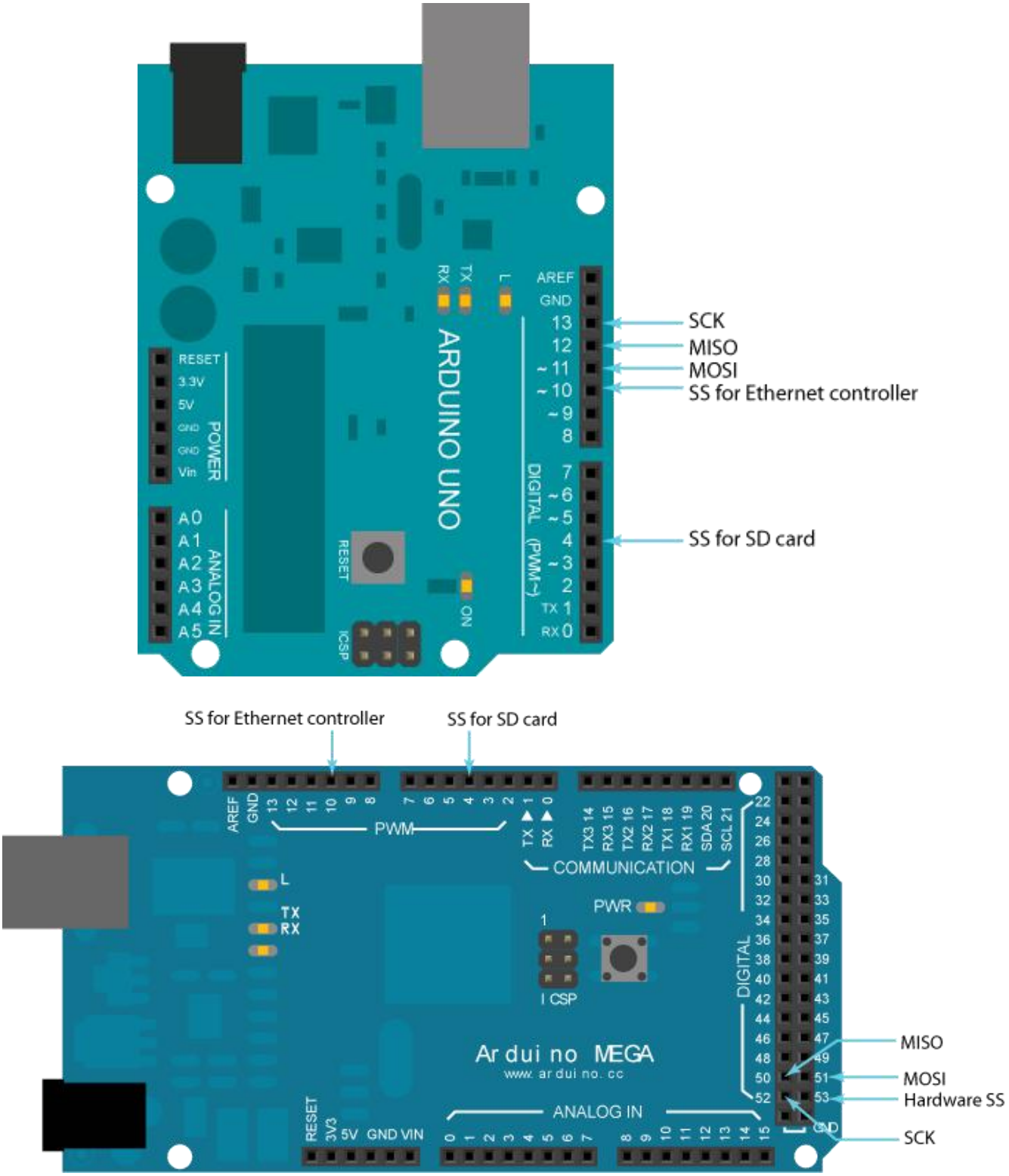
Examples

- Digital Potentiometer: Control an Analog Devices AD5171 Digital Potentiometer.
- Master Reader/Slave Writer: Program two Arduino boards to communicate with one another in a Master Reader/Slave Sender configuration via the I2C.
- Master Writer/Slave receiver:Program two Arduino boards to communicate with one another in a Master Writer/Slave Receiver configuration via the I2C.
- SFR Ranger Reader: Read an ultra-sonic range finder interfaced via the I2C.
- Add SerCom : Adding mores Serial interfaces to SAMD microcontrollers.

## Ethernet / Ethernet 2 library

These libraries are designed to work with the Arduino Ethernet Shield (Ethernet.h) or the Arduino Ethernet Shield 2 and Leonardo Ethernet (Ethernet2.h). The libraries are allow an Arduino board to connect to the internet. The board can serve as either a server accepting incoming connections or a client making outgoing ones. The libraries support up to four concurrent connection (incoming or outgoing or a combination). Ethernet library (Ethernet.h) manages the W5100 chip, while Ethernet2 library (Ethernet2.h) manages the W5500 chip; all the functions remain the same. Changing the library used allows to port the same code from Arduino Ethernet Shield to Arduino Ethernet 2 Shield or Arduino Leonardo Ethernet and vice versa.

Arduino communicates with the shield using the SPI bus. This is on digital pins 11, 12, and 13 on the Uno and pins 50, 51, and 52 on the Mega. On both boards, pin 10 is used as SS. On the Mega, the hardware SS pin, 53, is not used to select the W5100, but it must be kept as an output or the SPI interface won't work.




Examples

- ChatServer: set up a simple chat server.
- WebClient: make a HTTP request.

- WebClientRepeating: Make repeated HTTP requests.
- WebServer: host a simple HTML page that displays analog sensor values.
- BarometricPressureWebServer: outputs the values from a barometric pressure sensor as a web page.
- UDPSendReceiveString: Send and receive text strings via UDP.
- UdpNtpClient: Query a Network Time Protocol (NTP) server using UDP.
- DnsWebClient: DNS and DHCP-based Web client.
- DhcpChatServer: A simple DHCP Chat Server
- DhcpAddressPrinter: Get an IP address via DHCP and print it out
- TelnetClient: A simple Telnet client

## SD Library

The SD library allows for reading from and writing to SD cards, e.g. on the Arduino Ethernet Shield. It is built on sdfatlib by William Greiman. The library supports FAT16 and FAT32 file systems on standard SD cards and SDHC cards. It uses short 8.3 names for files. The file names passed to the SD library functions can include paths separated by forward-slashes, /, e.g. "directory/filename.txt". Because the working directory is always the root of the SD card, a name refers to the same file whether or not it includes a leading slash (e.g. "/file.txt" is equivalent to "file.txt"). As of version 1.0, the library supports opening multiple files.

The communication between the microcontroller and the SD card uses SPI, which takes place on digital pins 11, 12, and 13 (on most Arduino boards) or 50, 51, and 52 (Arduino Mega). Additionally, another pin must be used to select the SD card. This can be the hardware SS pin - pin 10 (on most Arduino boards) or pin 53 (on the Mega) - or another pin specified in the call to SD.begin(). Note that even if you don't use the hardware SS pin, it must be left as an output or the SD library won't work.

Notes on using the Library and various shields

Examples

- Card Info: Get info about your SD card.
- Datalogger: Log data from three analog sensors to an SD card.
- Dump File: Read a file from the SD card.
- Files: Create and destroy an SD card file.
- List Files: Print out the files in a directory on a SD card.
- Read Write: Read and write data to and from an SD card.

## Functions

## available()

### Description

available() gets the number of bytes available in the stream. This is only for bytes that have already arrived.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.available()

### Parameters

stream : an instance of a class that inherits from Stream.

### Returns

int : the number of bytes available to read

## read()

### Description

read() reads characters from an incoming stream to the buffer.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.read()

### Parameters

stream : an instance of a class that inherits from Stream.

### Returns

the first byte of incoming data available (or -1 if no data is available)

## flush()

### Description

flush() clears the buffer once all outgoing characters have been sent.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.flush()

### Parameters

stream : an instance of a class that inherits from Stream.

### Returns

boolean

## find()

### Description

find() reads data from the stream until the target string of given length is found The function returns true if target string is found, false if timed out.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.find(target)

### Parameters

stream : an instance of a class that inherits from Stream.
target : the string to search for (char)

### Returns

boolean

## findUntil()

### Description

findUntil() reads data from the stream until the target string of given length or terminator string is found.

The function returns true if target string is found, false if timed out

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.findUntil(target, terminal)

### Parameters

stream : an instance of a class that inherits from Stream.
target : the string to search for (char)
terminal : the terminal string in the search (char)

### Returns

boolean

## peek()

Read a byte from the file without advancing to the next one. That is, successive calls to peek() will return the same value, as will the next call to read().

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.peek()

### Parameters

stream : an instance of a class that inherits from Stream.

### Returns

The next byte (or character), or -1 if none is available.

## readBytes()

### Description

readBytes() read characters from a stream into a buffer. The function terminates if the determined length has been read, or it times out (see setTimeout()).

readBytes() returns the number of bytes placed in the buffer. A 0 means no valid data was found.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.readBytes(buffer, length)

### Parameters

stream : an instance of a class that inherits from Stream.
buffer: the buffer to store the bytes in (char[] or byte[])
length : the number of bytes to read (int)

### Returns

The number of bytes placed in the buffer

## readBytesUntil()

### Description

readBytesUntil() reads characters from a stream into a buffer. The function terminates if the terminator character is detected, the determined length has been read, or it times out (see setTimeout()).

readBytesUntil() returns the number of bytes placed in the buffer. A 0 means no valid data was found.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.readBytesUntil(character, buffer, length)

### Parameters

stream : an instance of a class that inherits from Stream.
character : the character to search for (char)
buffer: the buffer to store the bytes in (char[] or byte[])
length : the number of bytes to read (int)

### Returns

The number of bytes placed in the buffer

## readString()

### Description

readString() reads characters from a stream into a string. The function terminates if it times out (see setTimeout()).

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.readString()

### Parameters

none

### Returns

A string read from a stream

## readStringUntil()

### Description

readStringUntil() reads characters from a stream into a string. The function terminates if the terminator character is detected or it times out (see setTimeout()).

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.readString(terminator)

### Parameters

terminator : the character to search for (char)

### Returns

The entire string read from a stream, until the terminator character is detected

## parseInt()

### Description

parseInt() returns the first valid (long) integer number from the serial buffer. Characters that are not integers (or the minus sign) are skipped.

In particular:

- Initial characters that are not digits or a minus sign, are skipped;
- Parsing stops when no characters have been read for a configurable time-out value, or a non-digit is read;
- If no valid digits were read when the time-out (see Stream.setTimeout()) occurs, 0 is returned;

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.parseInt(list)

stream.parseInt(''list', char skipchar')

### Parameters

stream : an instance of a class that inherits from Stream.
list : the stream to check for ints (char) skipChar: used to skip the indicated char in the search. Used for example to skip thousands divider.

### Returns

long

## parseFloat()

### Description

parseFloat() returns the first valid floating point number from the current position. Initial characters that are not digits (or the minus sign) are skipped. parseFloat() is terminated by the first character that is not a floating point number.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.parseFloat(list)

### Parameters

stream : an instance of a class that inherits from Stream.
list : the stream to check for floats (char)

### Returns

Float

## setTimeout()

### Description

setTimeout() sets the maximum milliseconds to wait for stream data, it defaults to 1000 milliseconds. This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

### Syntax

stream.setTimeout(time)

### Parameters

stream : an instance of a class that inherits from Stream.
time : timeout duration in milliseconds (long).

### Parameters

None

# USB (32u4 based boards and Due/Zero only)

## Mouse and Keyboard libraries

These core libraries allow a 32u4 based boards or Due and Zero board to appear as a native Mouse and/or Keyboard to a connected computer.

A word of caution on using the Mouse and Keyboard libraries: if the Mouse or Keyboard library is constantly running, it will be difficult to program your board. Functions such as Mouse.move() and Keyboard.print() will move your cursor or send keystrokes to a connected computer and should only be called when you are ready to handle them. It is recommended to use a control system to turn this functionality on, like a physical switch or only responding to specific input you can control.

When using the Mouse or Keyboard library, it may be best to test your output first using Serial.print(). This way, you can be sure you know what values are being reported. Refer to the Mouse and Keyboard examples for some ways to handle this.

## Mouse

The mouse functions enable a Leonardo, Micro, or Due to control cursor movement on a connected computer. When updating the cursor position, it is always relative to the cursor's previous location.

- Mouse.begin()

### Description

Begins emulating the mouse connected to a computer. begin() must be called before controlling the computer. To end control, use Mouse.end().

### Syntax

Mouse.begin()

### Parameters

none

### Returns

nothing

### Example:

```
void setup(){
 pinMode(2, INPUT);
}

void loop(){
```

```
//initiate the Mouse library when button is pressed
if(digitalRead(2) == HIGH){
    Mouse.begin();
  }
```

- Mouse.click()

### Description

Sends a momentary click to the computer at the location of the cursor. This is the same as pressing and immediately releasing the mouse button.

Mouse.click() defaults to the left mouse button.

WARNING: When you use the Mouse.click() command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

### Syntax

Mouse.click();
Mouse.click(button);

### Parameters

button: which mouse button to press - char

- MOUSE_LEFT (default)
- MOUSE_RIGHT
- MOUSE_MIDDLE

### Returns

nothing

### Example

```
void setup(){
 pinMode(2,INPUT);
 //initiate the Mouse library
  Mouse.begin();
}

void loop(){
 //if the button is pressed, send a Right mouse click
 if(digitalRead(2) == HIGH){
    Mouse.click();
```

```
 }
}
```

- Mouse.end()

Description

Stops emulating the mouse connected to a computer. To start control, use Mouse.begin().

Syntax

Mouse.end()

Parameters

none

Returns

nothing

Example:

```
void setup(){
 pinMode(2,INPUT);
 //initiate the Mouse library
  Mouse.begin();
}

void loop(){
 //if the button is pressed, send a Right mouse click
 //then end the Mouse emulation
 if(digitalRead(2) == HIGH){
    Mouse.click();
    Mouse.end();
 }
}
```

- Mouse.move()

Description

Moves the cursor on a connected computer. The motion onscreen is always relative to the cursor's current location. Before using Mouse.move() you must call Mouse.begin()

WARNING: When you use the Mouse.move() command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

## Syntax

Mouse.move(xVal, yPos, wheel);

## Parameters

xVal: amount to move along the x-axis - *signed char*
yVal: amount to move along the y-axis - *signed char*
wheel: amount to move scroll wheel - *signed char*

## Returns

nothing

## Example:

```
const int xAxis = A1;          //analog sensor for X axis
const int yAxis = A2;          // analog sensor for Y axis

int range = 12;                // output range of X or Y movement
int responseDelay = 2;         // response delay of the mouse, in ms
int threshold = range/4;       // resting threshold
int center = range/2;          // resting position value
int minima[] = {
 1023, 1023};                  // actual analogRead minima for {x, y}
int maxima[] = {
 0,0};                         // actual analogRead maxima for {x, y}
int axis[] = {
  xAxis, yAxis};               // pin numbers for {x, y}
int mouseReading[2];           // final mouse readings for {x, y}
void setup() {
Mouse.begin();
}

void loop() {

// read and scale the two axes:
 int xReading = readAxis(0);
 int yReading = readAxis(1);

// move the mouse:
    Mouse.move(xReading, yReading, 0);
   delay(responseDelay);
}

/*
  reads an axis (0 or 1 for x or y) and scales the
```

```
  analog input range to a range from 0 to <range>
*/

int readAxis(int axisNumber) {
 int distance = 0;    // distance from center of the output range

 // read the analog input:
 int reading = analogRead(axis[axisNumber]);

// of the current reading exceeds the max or min for this axis,
// reset the max or min:
 if (reading < minima[axisNumber]) {
    minima[axisNumber] = reading;
 }
 if (reading > maxima[axisNumber]) {
    maxima[axisNumber] = reading;
 }

 // map the reading from the analog input range to the output range:
  reading = map(reading, minima[axisNumber], maxima[axisNumber], 0, range);

// if the output reading is outside from the
// rest position threshold,   use it:
 if (abs(reading - center) > threshold) {
    distance = (reading - center);
 }

 // the Y axis needs to be inverted in order to
 // map the movemment correctly:
 if (axisNumber == 1) {
    distance = -distance;
 }

 // return the distance for this axis:
 return distance;
}
```

- Mouse.press()

## Description

Sends a button press to a connected computer. A press is the equivalent of clicking and continuously holding the mouse button. A press is cancelled with Mouse.release().

Before using Mouse.press(), you need to start communication with Mouse.begin().

Mouse.press() defaults to a left button press.

WARNING: When you use the Mouse.press() command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

### Syntax

Mouse.press();
Mouse.press(button)

### Parameters

button: which mouse button to press - char

- MOUSE_LEFT (default)
- MOUSE_RIGHT
- MOUSE_MIDDLE

### Returns

nothing

### Example

```
void setup(){
 //The switch that will initiate the Mouse press
 pinMode(2,INPUT);
 //The switch that will terminate the Mouse press
 pinMode(3,INPUT);
 //initiate the Mouse library
  Mouse.begin();
}

void loop(){
 //if the switch attached to pin 2 is closed, press and hold the right mouse button
 if(digitalRead(2) == HIGH){
    Mouse.press();
 }
 //if the switch attached to pin 3 is closed, release the right mouse button
 if(digitalRead(3) == HIGH){
    Mouse.release();
 }
}
```

- Mouse.release()

## Description

Sends a message that a previously pressed button (invoked through Mouse.press()) is released. Mouse.release() defaults to the left button.

WARNING: When you use the Mouse.release() command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

## Syntax

Mouse.release();
Mouse.release(button);

## Parameters

button: which mouse button to press - char

- MOUSE_LEFT (default)
- MOUSE_RIGHT
- MOUSE_MIDDLE

## Returns

nothing

## Example

```
void setup(){
 //The switch that will initiate the Mouse press
 pinMode(2,INPUT);
 //The switch that will terminate the Mouse press
 pinMode(3,INPUT);
 //initiate the Mouse library
  Mouse.begin();
}

void loop(){
 //if the switch attached to pin 2 is closed, press and hold the right mouse button
 if(digitalRead(2) == HIGH){
    Mouse.press();
 }
 //if the switch attached to pin 3 is closed, release the right mouse button
 if(digitalRead(3) == HIGH){
    Mouse.release();
 }
}
```

- Mouse.isPressed()

### Description

Checks the current status of all mouse buttons, and reports if any are pressed or not.

### Syntax

Mouse.isPressed();
Mouse.isPressed(button);

### Parameters

When there is no value passed, it checks the status of the left mouse button.

button: which mouse button to check - char

- MOUSE_LEFT (default)
- MOUSE_RIGHT
- MOUSE_MIDDLE

### Returns

boolean : reports wether a button is pressed or not

### Example

```
void setup(){
 //The switch that will initiate the Mouse press
 pinMode(2,INPUT);
 //The switch that will terminate the Mouse press
 pinMode(3,INPUT);
 //Start serial communication with the computer
 Serial1.begin(9600);
 //initiate the Mouse library
  Mouse.begin();
}

void loop(){
 //a variable for checking the button's state
 int mouseState=0;
 //if the switch attached to pin 2 is closed, press and hold the right mouse button and save the
state ina   variable
 if(digitalRead(2) == HIGH){
    Mouse.press();
    mouseState=Mouse.isPressed();
 }
```

```
 //if the switch attached to pin 3 is closed, release the right mouse button and save the state in a variable
 if(digitalRead(3) == HIGH){
    Mouse.release();
    mouseState=Mouse.isPressed();
 }
 //print out the current mouse button state
 Serial1.println(mouseState);
 delay(10);
}
```

## Keyboard

The keyboard functions enable a Leonardo, Micro, or Due to send keystrokes to an attached computer.

Note: Not every possible ASCII character, particularly the non-printing ones, can be sent with the Keyboard library. The library supports the use of modifier keys. Modifier keys change the behavior of another key when pressed simultaneously. See here for additional information on supported keys and their use.

- Keyboard.begin()

### Description

When used with a Leonardo or Due board, Keyboard.begin() starts emulating a keyboard connected to a computer. To end control, use Keyboard.end().

### Syntax

Keyboard.begin()

### Parameters

none

### Returns

nothing

### Example

```
void setup() {
 // make pin 2 an input and turn on the
 // pullup resistor so it goes high unless
 // connected to ground:
 pinMode(2, INPUT_PULLUP);
```

```
  Keyboard.begin();
}

void loop() {
 //if the button is pressed
 if(digitalRead(2)==LOW){
    //Send the message
     Keyboard.print("Hello!");
 }
}
```

- Keyboard.end()

Description

Stops the keyboard emulation to a connected computer. To start keyboard emulation, use Keyboard.begin().

Syntax

Keyboard.end()

Parameters

none

Returns

nothing

Example

```
void setup() {
 //start keyboard communication
  Keyboard.begin();
 //send a keystroke
  Keyboard.print("Hello!");
 //end keyboard communication
  Keyboard.end();
}

void loop() {
 //do nothing
}
```

- Keyboard.press()

Description

When called, Keyboard.press() functions as if a key were pressed and held on your keyboard. Useful when using modifier keys. To end the key press, use Keyboard.release() or Keyboard.releaseAll().

It is necessary to call Keyboard.begin() before using press().

### Syntax

Keyboard.press()

### Parameters

char : the key to press

### Returns

None

### Example

```
// use this option for OSX:
char ctrlKey = KEY_LEFT_GUI;
// use this option for Windows and Linux:
//   char ctrlKey = KEY_LEFT_CTRL;

void setup() {
 // make pin 2 an input and turn on the
 // pullup resistor so it goes high unless
 // connected to ground:
 pinMode(2, INPUT_PULLUP);
 // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
 while (digitalRead(2) == HIGH) {
   // do nothing until pin 2 goes low
   delay(500);
 }
 delay(1000);
 // new document:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
 delay(100);
  Keyboard.releaseAll();
 // wait for new window to open:
```

```
 delay(1000);
}
```

- Keyboard.print()

### Description

Sends a keystroke to a connected computer.

Keyboard.print() must be called after initiating Keyboard.begin().

WARNING: When you use the Keyboard.print() command, the Arduino takes over your keyboard! Make sure you have control before you use the command. A pushbutton to toggle the keyboard control state is effective.

### Syntax

Keyboard.print(character)
Keyboard.print(characters)

### Parameters

character : a char or int to be sent to the computer as a keystroke characters : a string to be sent to the computer as a keystroke

### Returns

int : number of bytes sent

### Example

```
void setup() {
 // make pin 2 an input and turn on the
 // pullup resistor so it goes high unless
 // connected to ground:
 pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
 //if the button is pressed
 if(digitalRead(2)==LOW){
   //Send the message
    Keyboard.print("Hello!");
 }
}
```

- Keyboard.println()

## Description

Sends a keystroke to a connected computer, followed by a newline and carriage return.

Keyboard.println() must be called after initiating Keyboard.begin().

WARNING: When you use the Keyboard.println() command, the Arduino takes over your keyboard! Make sure you have control before you use the command. A pushbutton to toggle the keyboard control state is effective.

## Syntax

Keyboard.println()
Keyboard.println(character)
Keyboard.println(characters)

## Parameters

character : a char or int to be sent to the computer as a keystroke, followed by newline and carriage return.
characters : a string to be sent to the computer as a keystroke, followed by a newline and carriage return.

## Returns

int : number of bytes sent

## Example

```
void setup() {
 // make pin 2 an input and turn on the
 // pullup resistor so it goes high unless
 // connected to ground:
 pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
 //if the button is pressed
 if(digitalRead(2)==LOW){
   //Send the message
    Keyboard.println("Hello!");
 }
}
```

- Keyboard.release()

## Description

Sends a keystroke to a connected computer, followed by a newline and carriage return.

Keyboard.println() must be called after initiating Keyboard.begin().

WARNING: When you use the Keyboard.println() command, the Arduino takes over your keyboard! Make sure you have control before you use the command. A pushbutton to toggle the keyboard control state is effective.

## Syntax

Keyboard.println()
Keyboard.println(character)
Keyboard.println(characters)

## Parameters

character : a char or int to be sent to the computer as a keystroke, followed by newline and carriage return.
characters : a string to be sent to the computer as a keystroke, followed by a newline and carriage return.

## Returns

int : number of bytes sent

## Example

```
void setup() {
 // make pin 2 an input and turn on the
 // pullup resistor so it goes high unless
 // connected to ground:
 pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
 //if the button is pressed
 if(digitalRead(2)==LOW){
   //Send the message
    Keyboard.println("Hello!");
 }
}
```

- Keyboard.releaseAll()

## Description

Lets go of all keys currently pressed. See Keyboard.press() for additional information.

## Syntax

Keyboard.releaseAll()

## Parameters

None

## Returns

int : the number of keys released

## Example

```
// use this option for OSX:
char ctrlKey = KEY_LEFT_GUI;
// use this option for Windows and Linux:
//   char ctrlKey = KEY_LEFT_CTRL;

void setup() {
 // make pin 2 an input and turn on the
 // pullup resistor so it goes high unless
 // connected to ground:
 pinMode(2, INPUT_PULLUP);
 // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
 while (digitalRead(2) == HIGH) {
   // do nothing until pin 2 goes low
   delay(500);
 }
 delay(1000);
 // new document:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
 delay(100);
  Keyboard.releaseAll();
 // wait for new window to open:
 delay(1000);
}
```

- Keyboard.write()

### Description

Sends a keystroke to a connected computer. This is similar to pressing and releasing a key on your keyboard. You can send some ASCII characters or the additional keyboard modifiers and special keys.

Only ASCII characters that are on the keyboard are supported. For example, ASCII 8 (backspace) would work, but ASCII 25 (Substitution) would not. When sending capital letters, Keyboard.write() sends a shift command plus the desired character, just as if typing on a keyboard. If sending a numeric type, it sends it as an ASCII character (ex. Keyboard.write(97) will send 'a').

For a complete list of ASCII characters, see ASCIITable.com.

WARNING: When you use the Keyboard.write() command, the Arduino takes over your keyboard! Make sure you have control before you use the command. A pushbutton to toggle the keyboard control state is effective.

### Syntax

Keyboard.write(character)

### Parameters

character : a char or int to be sent to the computer. Can be sent in any notation that's acceptable for a char. For example, all of the below are acceptable and send the same value, 65 or ASCII A:

```
Keyboard.write(65);          // sends ASCII value 65, or A
Keyboard.write('A');             // same thing as a quoted character
Keyboard.write(0x41);        // same thing in hexadecimal
Keyboard.write(0b01000001); // same thing in binary (weird choice, but it works)
```

### Returns

int : number of bytes sent

### Example

```
void setup() {
 // make pin 2 an input and turn on the
 // pullup resistor so it goes high unless
 // connected to ground:
 pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}
```

```
void loop() {
 //if the button is pressed
 if(digitalRead(2)==LOW){
   //Send an ASCII 'A',
    Keyboard.write(65);
 }
}
```

## Examples

- KeyboardAndMouseControl: Demonstrates the Mouse and Keyboard commands in one program.

This example illustrates the use of the Mouse and Keyboard libraries together. Five momentary switches act as directional buttons for your cursor. When a button is pressed, the cursor on your screen will move, and a keypress, corresponding to the letter associated with the direction, will be sent to the computer. Once you have the Leonardo, Micro or Due programmed and wired up, open up your favorite text editor to see the results.

NB: When you use the Mouse and Keyboard library functions, the Arduino takes over your computer's cursor! To insure you don't lose control of your computer while running a sketch with this function, make sure to set up a controller before you call Mouse.move().

## Hardware Required

- Arduino Leonardo, Micro or Arduino Due board
- 5 pushbuttons
- 5 10k ohm resistors
- hook-up wires
- breadboard

## Software Required

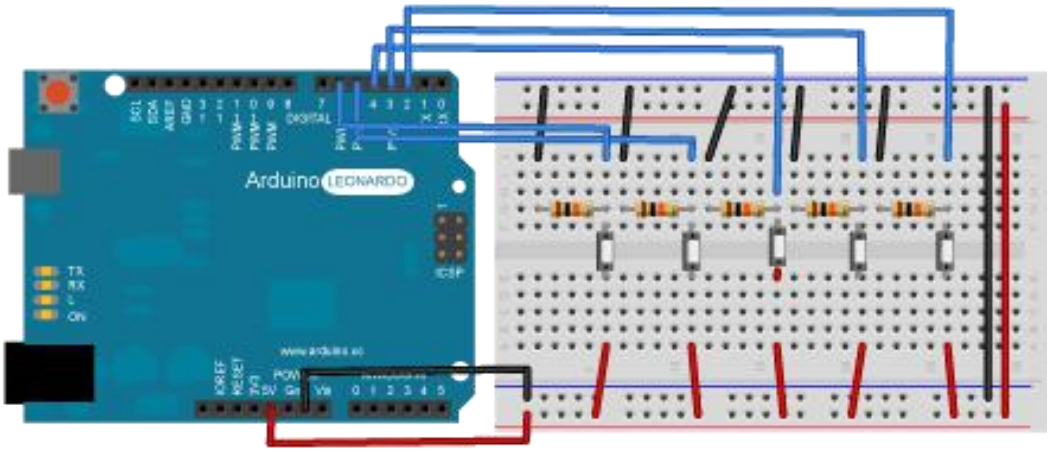
- Any text editor

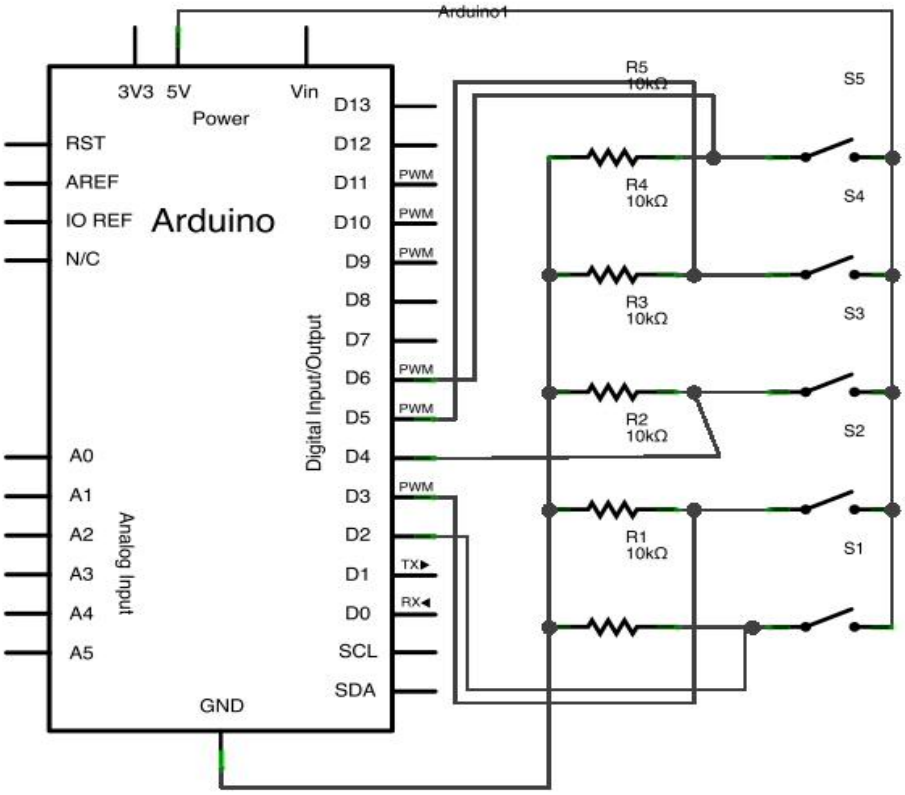
## Circuit

Attach one end of the the pushbuttons to pins 2, 3, 4, 5, and 6 on the board. Attach the other end to +5V. Use the resistors as pull-downs, providing a reference to ground for the switches. Attach them from the pin connecting to the board to ground.

Once you've programmed your board, unplug the USB cable and open a text editor. Connect your board to your computer and press the buttons to write in the document as you move the cursor.

## Schematic




click the images to enlarge

Code

```
/*
  KeyboardAndMouseControl
```

```
 Controls the mouse from five pushbuttons on an Arduino Leonardo, Micro or Due.

 Hardware:
 * 5 pushbuttons attached to D2, D3, D4, D5, D6

 The mouse movement is always relative. This sketch reads
 four pushbuttons, and uses them to set the movement of the mouse.

 WARNING:  When you use the Mouse.move() command, the Arduino takes
 over your mouse!  Make sure you have control before you use the mouse commands.

 created 15 Mar 2012
 modified 27 Mar 2012
 by Tom Igoe

 this code is in the public domain

 */

#include "Keyboard.h"
#include "Mouse.h"

// set pin numbers for the five buttons:
const int upButton = 2;
const int downButton = 3;
const int leftButton = 4;
const int rightButton = 5;
const int mouseButton = 6;

void setup() { // initialize the buttons' inputs:
  pinMode(upButton, INPUT);
  pinMode(downButton, INPUT);
  pinMode(leftButton, INPUT);
  pinMode(rightButton, INPUT);
  pinMode(mouseButton, INPUT);

  Serial.begin(9600);
  // initialize mouse control:
   Mouse.begin();
   Keyboard.begin();
}

void loop() {
```

```
// use serial input to control the mouse:
if (Serial.available() > 0) {
  char inChar = Serial.read();

  switch (inChar) {
    case 'u':
      // move mouse up
       Mouse.move(0, -40);
      break;
    case 'd':
      // move mouse down
       Mouse.move(0, 40);
      break;
    case 'l':
      // move mouse left
       Mouse.move(-40, 0);
      break;
    case 'r':
      // move mouse right
       Mouse.move(40, 0);
      break;
    case 'm':
      // perform mouse left click
       Mouse.click(MOUSE_LEFT);
      break;
  }
}

// use the pushbuttons to control the keyboard:
if (digitalRead(upButton) == HIGH) {
   Keyboard.write('u');
}
if (digitalRead(downButton) == HIGH) {
   Keyboard.write('d');
}
if (digitalRead(leftButton) == HIGH) {
   Keyboard.write('l');
}
if (digitalRead(rightButton) == HIGH) {
   Keyboard.write('r');
}
if (digitalRead(mouseButton) == HIGH) {
   Keyboard.write('m');
```

```
 }
}
```

- KeyboardMessage: Sends a text string when a button is pressed.

When the button is pressed in this example, a text string is sent to the computer as keyboard input. The string reports the number of times the button has been pressed. Once you have the Leonardo programmed and wired up, open up your favourite text editor to see the results.

NB: When you use the Keyboard.print() command, the Arduino takes over your computer's keyboard! To insure you don't lose control of your computer while running a sketch with this function, make sure to set up a reliable control system before you call Keyboard.print(). This sketch includes a pushbutton to toggle the keyboard, so that it only runs after the button is pressed.

### Hardware Required

- Arduino Leonardo, Micro, or Due board
- momentary pushbutton
- 10k ohm resistor

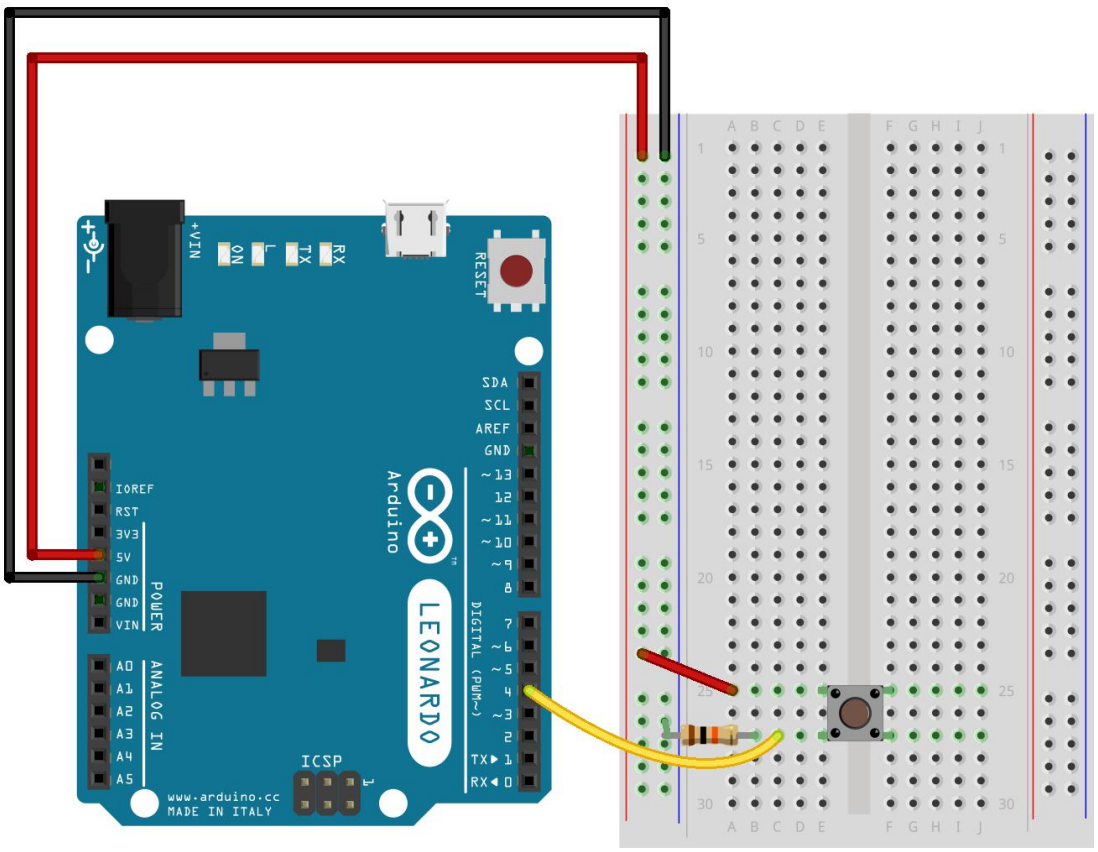
### Software Required

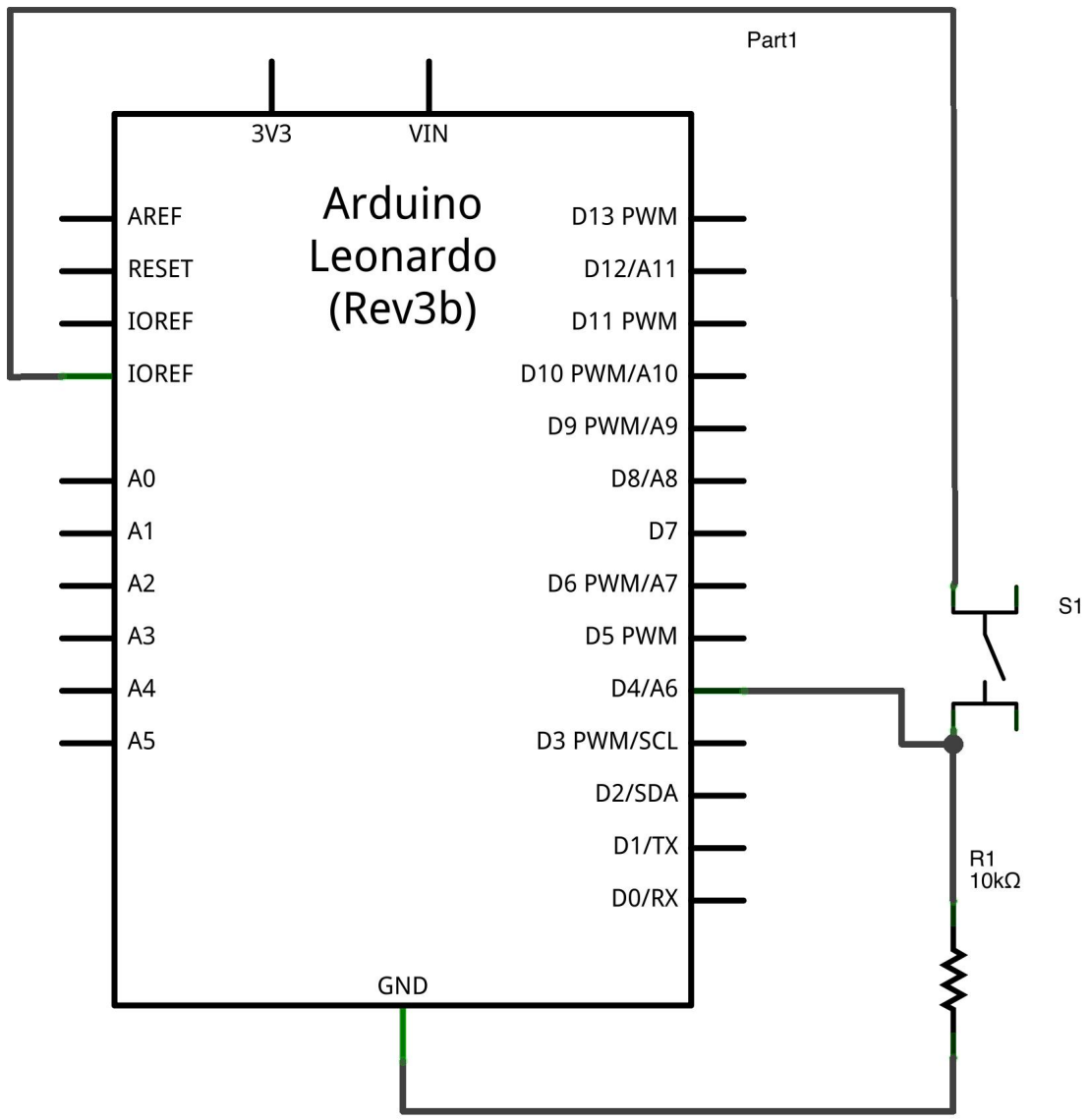
- Any text editor

### Circuit

Attach one pin of the pushbutton to pin 4 on the Arduino. Attach the other pin to 5V. Use the resistor as a pull-down, providing a reference to ground, by attaching it from pin 4 to ground.

Once you've programmed your board, unplug the USB cable, open a text editor and put the text cursor at in the typing area. Connect the board to your computer through USB again and press the button to write in the document.


+VIN
ON
L
TX
RX
RESET
SDA
SCL
AREF
GND
~13
12
~11
~10
~9
8
7
~6
~5
4
~3
2
TX▶1
RX◀0
DIGITAL (PWM~)
IOREF
RST
3V3
5V
GND
GND
VIN
POWER
A0
A1
A2
A3
A4
A5
ANALOG IN
ICSP
Arduino
LEONARDO
www.arduino.cc
MADE IN ITALY
A B C D E
F G H I J
1
5
10
15
20
25
30




Code

```
/*
 Keyboard Message test

 For the Arduino Leonardo and Micro.

 Sends a text string when a button is pressed.

 The circuit:
 * pushbutton attached from pin 4 to +5V
 * 10-kilohm resistor attached from pin 4 to ground

 created 24 Oct 2011
 modified 27 Mar 2012
```

```
 by Tom Igoe
 modified 11 Nov 2013
 by Scott Fitzgerald

 This example code is in the public domain.

 http://www.arduino.cc/en/Tutorial/KeyboardMessage
 */

#include "Keyboard.h"

const int buttonPin = 4;              // input pin for pushbutton
int previousButtonState = HIGH;   // for checking the state of a pushButton
int counter = 0;                          // button push counter

void setup() {
 // make the pushButton pin an input:
 pinMode(buttonPin, INPUT);
 // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
 // read the pushbutton:
 int buttonState = digitalRead(buttonPin);
 // if the button state has changed,
 if ((buttonState != previousButtonState)
       // and it's currently pressed:
       && (buttonState == HIGH)) {
    // increment the button counter
     counter++;
    // type out a message
     Keyboard.print("You pressed the button ");
     Keyboard.print(counter);
     Keyboard.println(" times.");
 }
 // save the current button state for comparison next time:
  previousButtonState = buttonState;
}
```

- KeyboardLogout : Logs out the current user with key commands

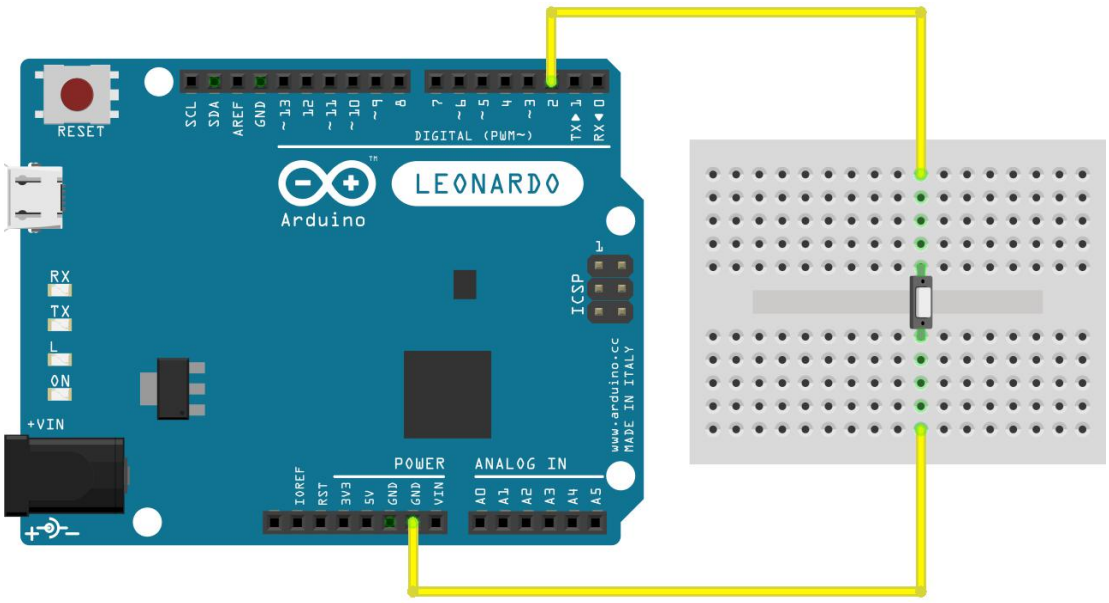
This example uses the Keyboard library to log you out of your user session on your computer when pin 2 on your Leonardo, Micro or Due is pulled to ground. The sketch simulates the keypress in sequence of two or three keys at the same time and after a short delay it releases them.

NB: When you use the Keyboard.print() command, the Arduino takes over your computer's keyboard! To insure you don't lose control of your computer while running a sketch with this function, make sure to set up a reliable control system before you call Keyboard.print(). This sketch is designed to only send a Keyboard command after a pin has been pulled to ground.

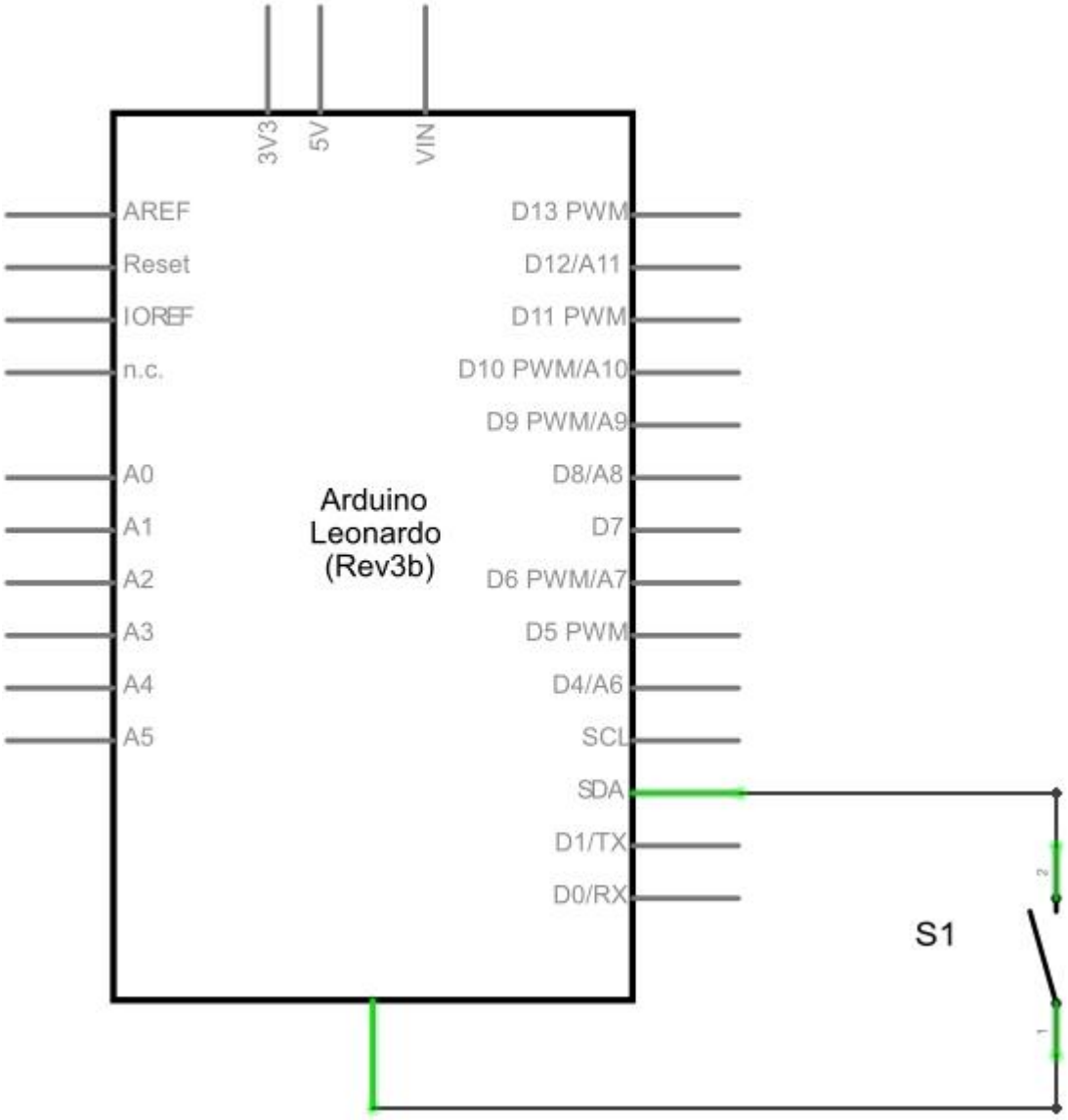
## Hardware Required

- Arduino Leonardo, Micro, or Due board
- pushbutton
- hook-up wires
- breadboard

## Circuit




## Schematic




## Code

Before you upload the program to your board, make sure to assign the correct OS you are currently using to the platform variable.

While the sketch is running, pressing the button will connect pin 2 to ground and the board will send the logout sequence to the USB connected pc.

```
/*
  Keyboard logout

 This sketch demonstrates the Keyboard library.

 When you connect pin 2 to ground, it performs a logout.
 It uses keyboard combinations to do this, as follows:
```

```
 On Windows, CTRL-ALT-DEL followed by ALT-l
 On Ubuntu, CTRL-ALT-DEL, and ENTER
 On OSX, CMD-SHIFT-q

 To wake: Spacebar.

 Circuit:
 * Arduino Leonardo or Micro
 * wire to connect D2 to ground.

 created 6 Mar 2012
 modified 27 Mar 2012
 by Tom Igoe

 This example is in the public domain

 http://www.arduino.cc/en/Tutorial/KeyboardLogout
 */

#define OSX 0
#define WINDOWS 1
#define UBUNTU 2

#include "Keyboard.h"

// change this to match your platform:
int platform = OSX;

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
   Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
     // do nothing until pin 2 goes low
     delay(500);
  }
  delay(1000);
```

```
switch (platform) {
  case OSX:
    Keyboard.press(KEY_LEFT_GUI);
    // Shift-Q logs out:
    Keyboard.press(KEY_LEFT_SHIFT);
    Keyboard.press('Q');
    delay(100);
    Keyboard.releaseAll();
    // enter:
    Keyboard.write(KEY_RETURN);
    break;
  case WINDOWS:
    // CTRL-ALT-DEL:
    Keyboard.press(KEY_LEFT_CTRL);
    Keyboard.press(KEY_LEFT_ALT);
    Keyboard.press(KEY_DELETE);
    delay(100);
    Keyboard.releaseAll();
    //ALT-I:
    delay(2000);
    Keyboard.press(KEY_LEFT_ALT);
    Keyboard.press('I');
    Keyboard.releaseAll();
    break;
  case UBUNTU:
    // CTRL-ALT-DEL:
    Keyboard.press(KEY_LEFT_CTRL);
    Keyboard.press(KEY_LEFT_ALT);
    Keyboard.press(KEY_DELETE);
    delay(1000);
    Keyboard.releaseAll();
    // Enter to confirm logout:
    Keyboard.write(KEY_RETURN);
    break;
}

// do nothing:
while (true);
}
```

- KeyboardSerial: Reads a byte from the serial port, and sends back a keystroke.

This example listens for a byte coming from the serial port. When received, the board sends a keystroke back to the computer. The sent keystroke is one higher than what is received, so if you send an "a" from the serial monitor, you'll receive a "b" from the board connected to the computer. A "1" will return a "2" and so on.

NB: When you use the Keyboard.print() command, the Leonardo, Micro or Due board takes over your computer's keyboard! To insure you don't lose control of your computer while running a sketch with this function, make sure to set up a reliable control system before you call Keyboard.print(). This sketch is designed to only send a Keyboard command after the board has received a byte over the serial port.

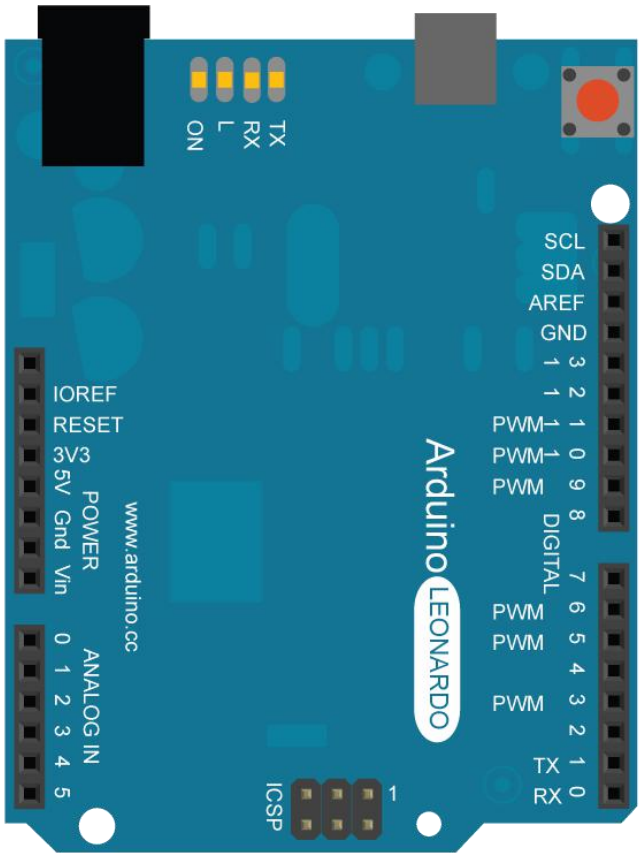
## Hardware Required

- Arduino Leonardo, Micro, or Due board

## Circuit

Connect your board to your computer with a micro-USB cable.

Once programmed, open your serial monitor and send a byte. The board will reply with a keystroke that is one number higher.




## Code

```
/*
 Keyboard test
```

```
 For the Arduino Leonardo, Micro or Due

 Reads a byte from the serial port, sends a keystroke back.
 The sent keystroke is one higher than what's received, e.g.
 if you send a, you get b, send A you get B, and so forth.

 The circuit:
 * none

 created 21 Oct 2011
 modified 27 Mar 2012
 by Tom Igoe

 This example code is in the public domain.

 http://www.arduino.cc/en/Tutorial/KeyboardSerial
 */

#include "Keyboard.h"

void setup() {
 // open the serial port:
 Serial.begin(9600);
 // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
 // check for incoming serial data:
 if (Serial.available() > 0) {
   // read incoming serial data:
   char inChar = Serial.read();
   // Type the next ASCII value from what you received:
    Keyboard.write(inChar + 1);
 }
}
```

- KeyboardReprogram : opens a new window in the Arduino IDE and reprograms the board with a simple blink program

This example uses the Keyboard library to open a new Arduino Software (IDE) sketch window, send keyboard commands that type in the Blink example, and reprograms the board. After

running this sketch and connecting pin 2 to ground using the pushbutton, the board will have a new program, Blink.

NB: When you use the Keyboard.print() command, the Arduino takes over your computer's keyboard! To insure you don't lose control of your computer while running a sketch with this function, make sure to set up a reliable control system before you call Keyboard.print(). This sketch is designed to only send Keyboard commands after digital pin 2 is pulled to ground.

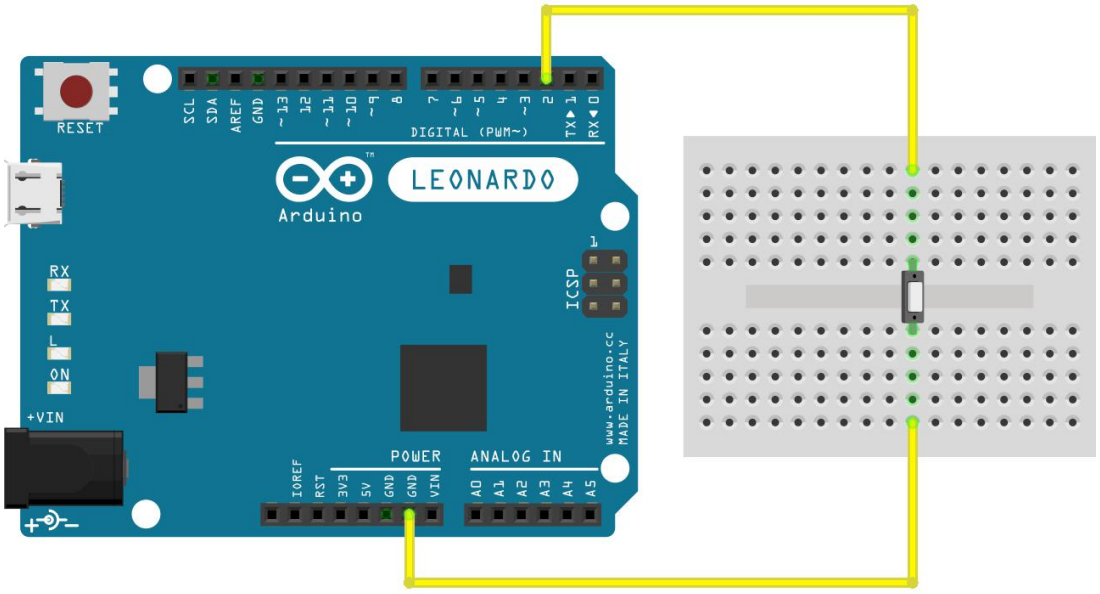
## Hardware Required

- Arduino Leonardo, Micro, or Due board
- pushbutton
- hook-up wires
- breadboard

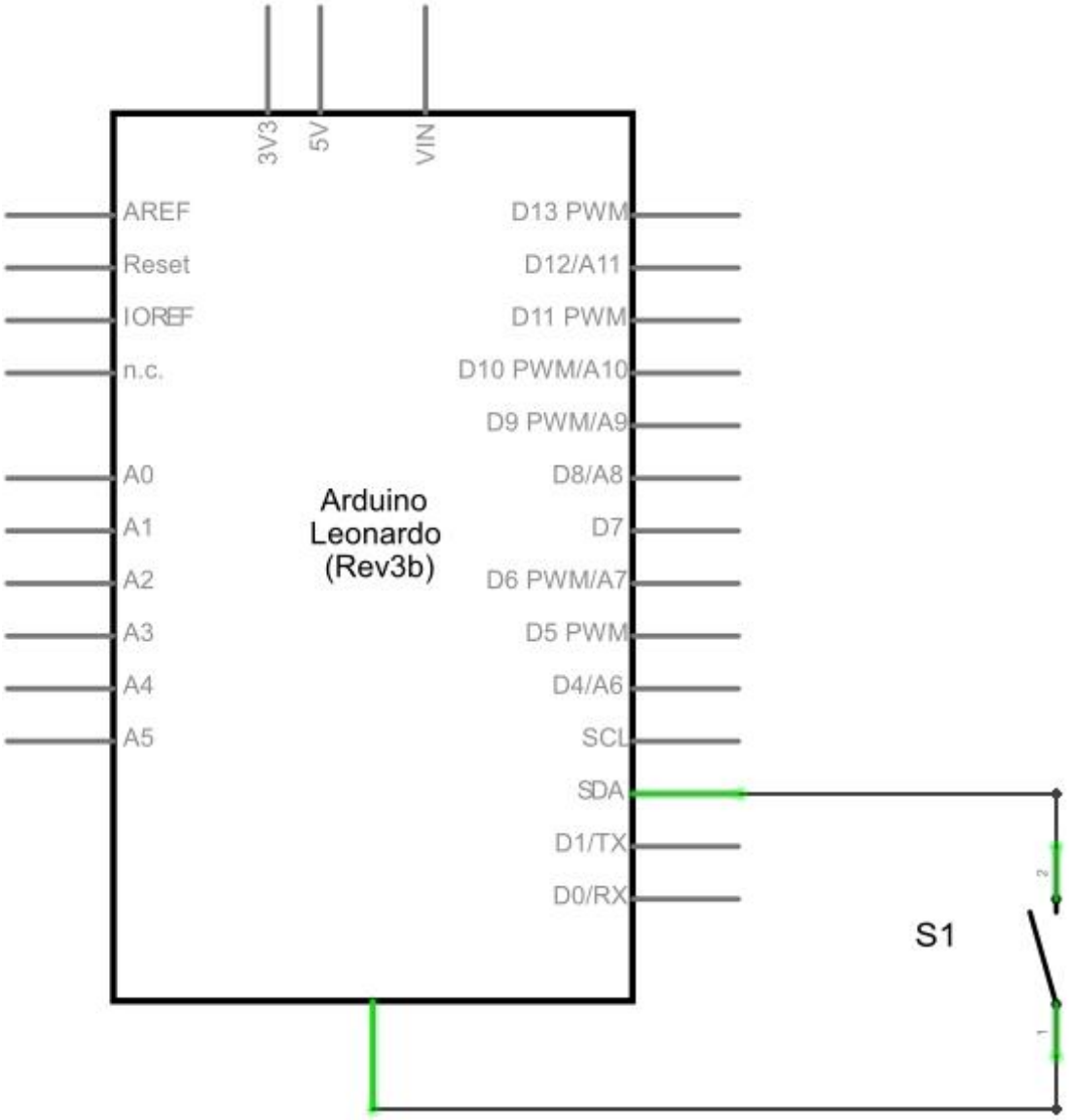
## Software Required

- Arduino IDE running

## Circuit




## Schematic




## Code

Connect your board to the USB port, then push the button to connect D2 with GND and initiate the sketch keyboard keypress emulation. Remember to have the Arduino Software (IDE) window selected before you press the button.

```
/*
  Arduino Programs Blink

This sketch demonstrates the Keyboard library.

For Leonardo and Due boards only.

When you connect pin 2 to ground, it creates a new
window with a key combination (CTRL-N),
```

```
 then types in the Blink sketch, then auto-formats the text
 using another key combination (CTRL-T), then
 uploads the sketch to the currently selected Arduino using
 a final key combination (CTRL-U).

 Circuit:
 * Arduino Leonardo, Micro, Due, LilyPad USB, or Yún
 * wire to connect D2 to ground.

 created 5 Mar 2012
 modified 29 Mar 2012
 by Tom Igoe
 modified 3 May 2014
 by Scott Fitzgerald

 This example is in the public domain

 http://www.arduino.cc/en/Tutorial/KeyboardReprogram
 */

#include "Keyboard.h"

// use this option for OSX.
// Comment it out if using Windows or Linux:
char ctrlKey = KEY_LEFT_GUI;
// use this option for Windows and Linux.
// leave commented out if using OSX:
//   char ctrlKey = KEY_LEFT_CTRL;


void setup() {
 // make pin 2 an input and turn on the
 // pullup resistor so it goes high unless
 // connected to ground:
 pinMode(2, INPUT_PULLUP);
 // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
 while (digitalRead(2) == HIGH) {
   // do nothing until pin 2 goes low
   delay(500);
 }
```

```
delay(1000);
// new document:
 Keyboard.press(ctrlKey);
 Keyboard.press('n');
delay(100);
 Keyboard.releaseAll();
// wait for new window to open:
delay(1000);

// versions of the Arduino IDE after 1.5 pre-populate
// new sketches with setup() and loop() functions
// let's clear the window before typing anything new
//   select all
 Keyboard.press(ctrlKey);
 Keyboard.press('a');
delay(500);
 Keyboard.releaseAll();
// delete the selected text
 Keyboard.write(KEY_BACKSPACE);
delay(500);

// Type out "blink":
 Keyboard.println("void setup() {");
 Keyboard.println("pinMode(13, OUTPUT);");
 Keyboard.println("}");
 Keyboard.println();
 Keyboard.println("void loop() {");
 Keyboard.println("digitalWrite(13, HIGH);");
 Keyboard.print("delay(3000);");
// 3000 ms is too long. Delete it:
for (int keystrokes = 0; keystrokes < 6; keystrokes++) {
   delay(500);
    Keyboard.write(KEY_BACKSPACE);
}
// make it 1000 instead:
 Keyboard.println("1000);");
 Keyboard.println("digitalWrite(13, LOW);");
 Keyboard.println("delay(1000);");
 Keyboard.println("}");
// tidy up:
 Keyboard.press(ctrlKey);
 Keyboard.press('t');
delay(100);
 Keyboard.releaseAll();
```

```
 delay(3000);
 // upload code:
  Keyboard.press(ctrlKey);
  Keyboard.press('u');
 delay(100);
  Keyboard.releaseAll();

 // wait for the sweet oblivion of reprogramming:
 while (true);
}
```

- ButtonMouseControl: Control cursor movement with 5 pushbuttons.

This example uses the Keyboard library to open a new Arduino Software (IDE) sketch window, send keyboard commands that type in the Blink example, and reprograms the board. After running this sketch and connecting pin 2 to ground using the pushbutton, the board will have a new program, Blink.

NB: When you use the Keyboard.print() command, the Arduino takes over your computer's keyboard! To insure you don't lose control of your computer while running a sketch with this function, make sure to set up a reliable control system before you call Keyboard.print(). This sketch is designed to only send Keyboard commands after digital pin 2 is pulled to ground.

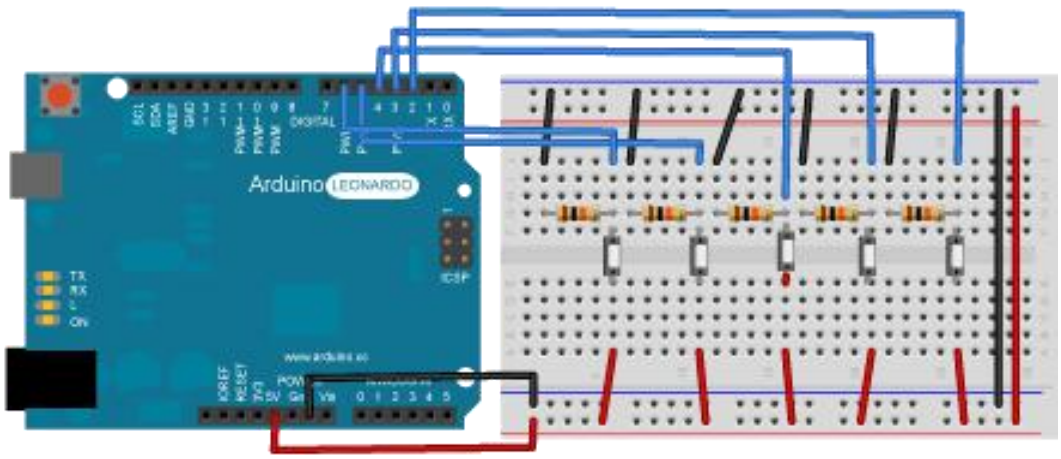
### Hardware Required

- Arduino Leonardo, Micro, or Due board
- pushbutton
- hook-up wires
- breadboard

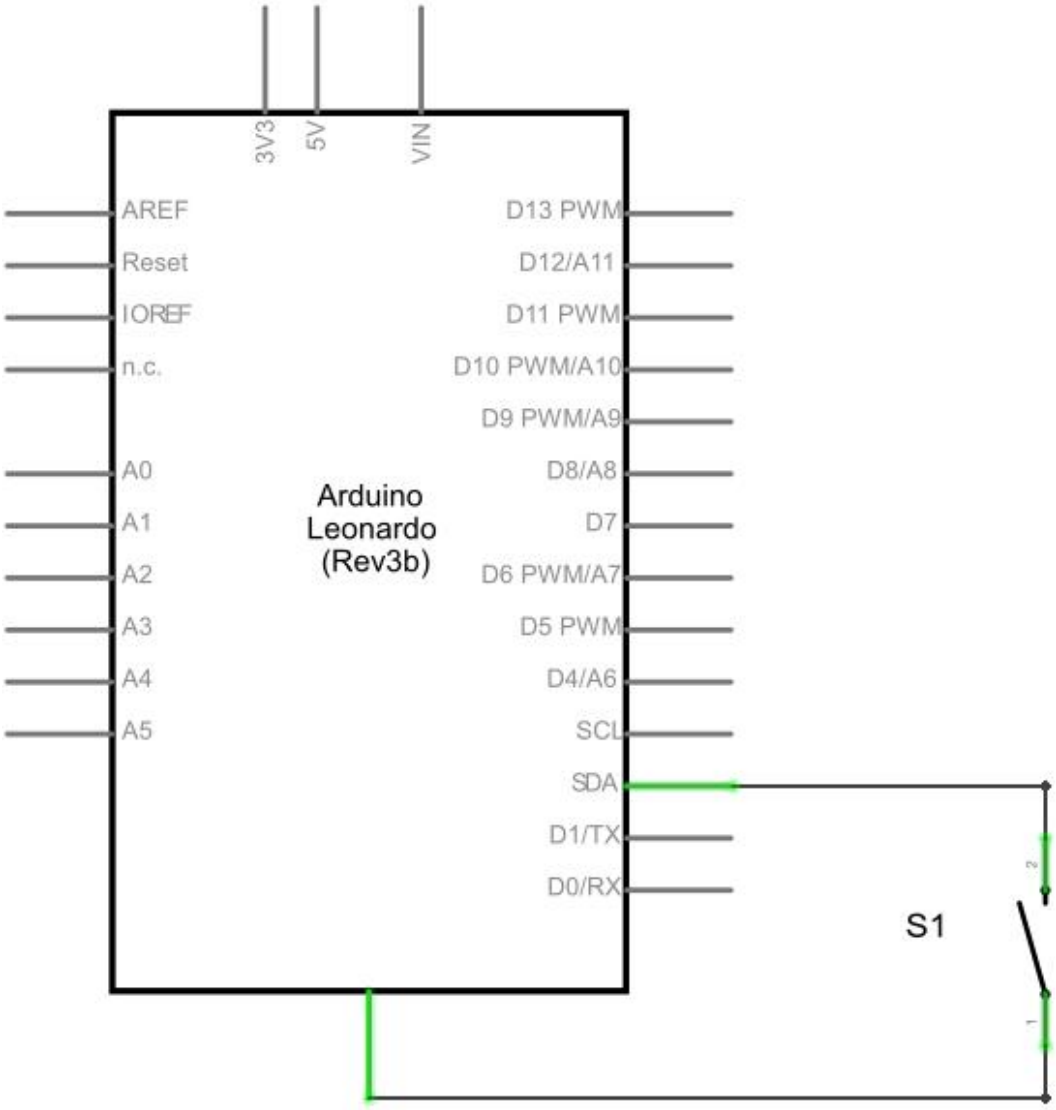
### Software Required

- Arduino IDE running

### Circuit



## Schematic




## Code

Connect your board to the USB port, then push the button to connect D2 with GND and initiate the sketch keyboard keypress emulation. Remember to have the Arduino Software (IDE) window selected before you press the button.

```
/*
  Arduino Programs Blink

This sketch demonstrates the Keyboard library.

For Leonardo and Due boards only.

When you connect pin 2 to ground, it creates a new
window with a key combination (CTRL-N),
```

```
 then types in the Blink sketch, then auto-formats the text
 using another key combination (CTRL-T), then
 uploads the sketch to the currently selected Arduino using
 a final key combination (CTRL-U).

 Circuit:
 * Arduino Leonardo, Micro, Due, LilyPad USB, or Yún
 * wire to connect D2 to ground.

 created 5 Mar 2012
 modified 29 Mar 2012
 by Tom Igoe
 modified 3 May 2014
 by Scott Fitzgerald

 This example is in the public domain

 http://www.arduino.cc/en/Tutorial/KeyboardReprogram
 */

#include "Keyboard.h"

// use this option for OSX.
// Comment it out if using Windows or Linux:
char ctrlKey = KEY_LEFT_GUI;
// use this option for Windows and Linux.
// leave commented out if using OSX:
//   char ctrlKey = KEY_LEFT_CTRL;


void setup() {
 // make pin 2 an input and turn on the
 // pullup resistor so it goes high unless
 // connected to ground:
 pinMode(2, INPUT_PULLUP);
 // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
 while (digitalRead(2) == HIGH) {
   // do nothing until pin 2 goes low
   delay(500);
 }
```

```
delay(1000);
// new document:
 Keyboard.press(ctrlKey);
 Keyboard.press('n');
delay(100);
 Keyboard.releaseAll();
// wait for new window to open:
delay(1000);

// versions of the Arduino IDE after 1.5 pre-populate
// new sketches with setup() and loop() functions
// let's clear the window before typing anything new
//   select all
 Keyboard.press(ctrlKey);
 Keyboard.press('a');
delay(500);
 Keyboard.releaseAll();
// delete the selected text
 Keyboard.write(KEY_BACKSPACE);
delay(500);

// Type out "blink":
 Keyboard.println("void setup() {");
 Keyboard.println("pinMode(13, OUTPUT);");
 Keyboard.println("}");
 Keyboard.println();
 Keyboard.println("void loop() {");
 Keyboard.println("digitalWrite(13, HIGH);");
 Keyboard.print("delay(3000);");
// 3000 ms is too long. Delete it:
for (int keystrokes = 0; keystrokes < 6; keystrokes++) {
   delay(500);
    Keyboard.write(KEY_BACKSPACE);
}
// make it 1000 instead:
 Keyboard.println("1000);");
 Keyboard.println("digitalWrite(13, LOW);");
 Keyboard.println("delay(1000);");
 Keyboard.println("}");
// tidy up:
 Keyboard.press(ctrlKey);
 Keyboard.press('t');
delay(100);
 Keyboard.releaseAll();
```

```
 delay(3000);
 // upload code:
  Keyboard.press(ctrlKey);
  Keyboard.press('u');
 delay(100);
  Keyboard.releaseAll();

 // wait for the sweet oblivion of reprogramming:
 while (true);
}
```

- JoystickMouseControl: Controls a computer's cursor movement with a Joystick when a button is pressed.

Using the Mouse library, you can controls a computer's onscreen cursor with an Arduino Leonardo, Micro, or Due. This particular example uses a pushbutton to turn on and off mouse control with a joystick.

Cursor movement from the Arduino is always relative. So every time the analog input is read, the cursor's position is updated relative to it's current position.

Two analog inputs ranging from 0 to 1023 are translated to ranges of -12 to 12. The sketch assumes that the joystick resting values are around the middle of the range, but that they vary within a threshold.

The pushbutton allows you to toggle mouse control on and off. As an option you may connect a status LED to pin 5 that lights upwhen the Arduino is controlling the mouse. A second pushbutton may be connected with another 10k ohm pulldown (to GND) resistor to D3 to act as the left click of the mouse.

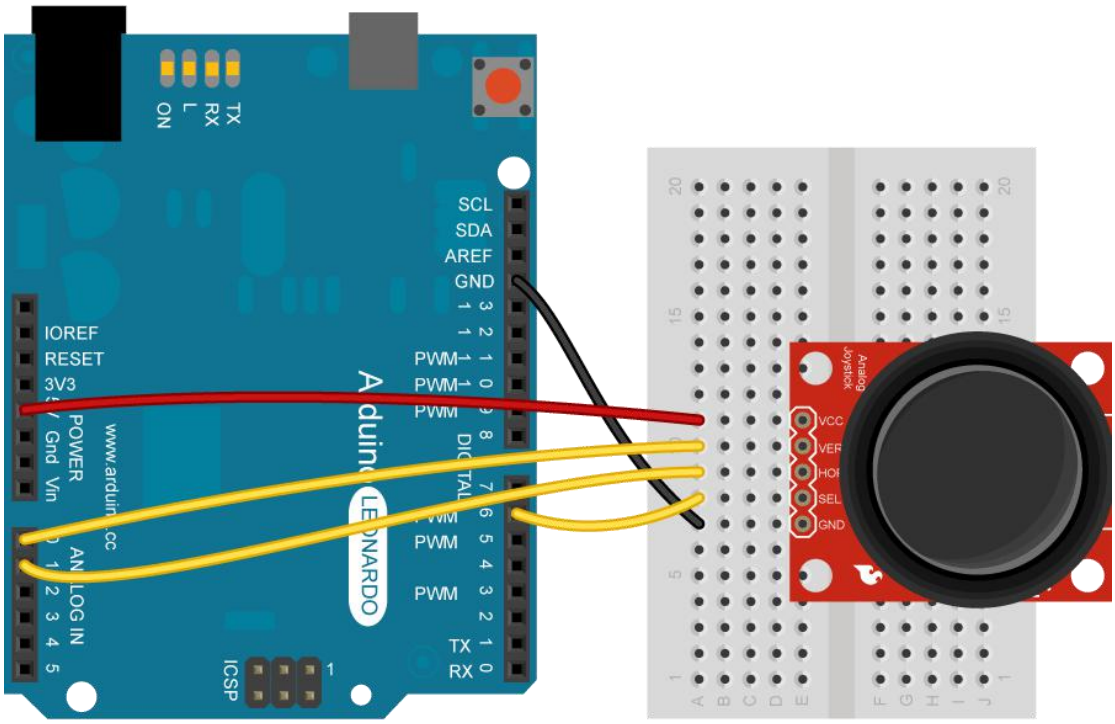
NB: When you use the Mouse.move() command, the Arduino takes over your computer's cursor! To insure you don't lose control of your computer while running a sketch with this function, make sure to set up a controller before you call Mouse.move(). This sketch includes a pushbutton to toggle the mouse control state, so you can turn on and off mouse control.

### Hardware Required

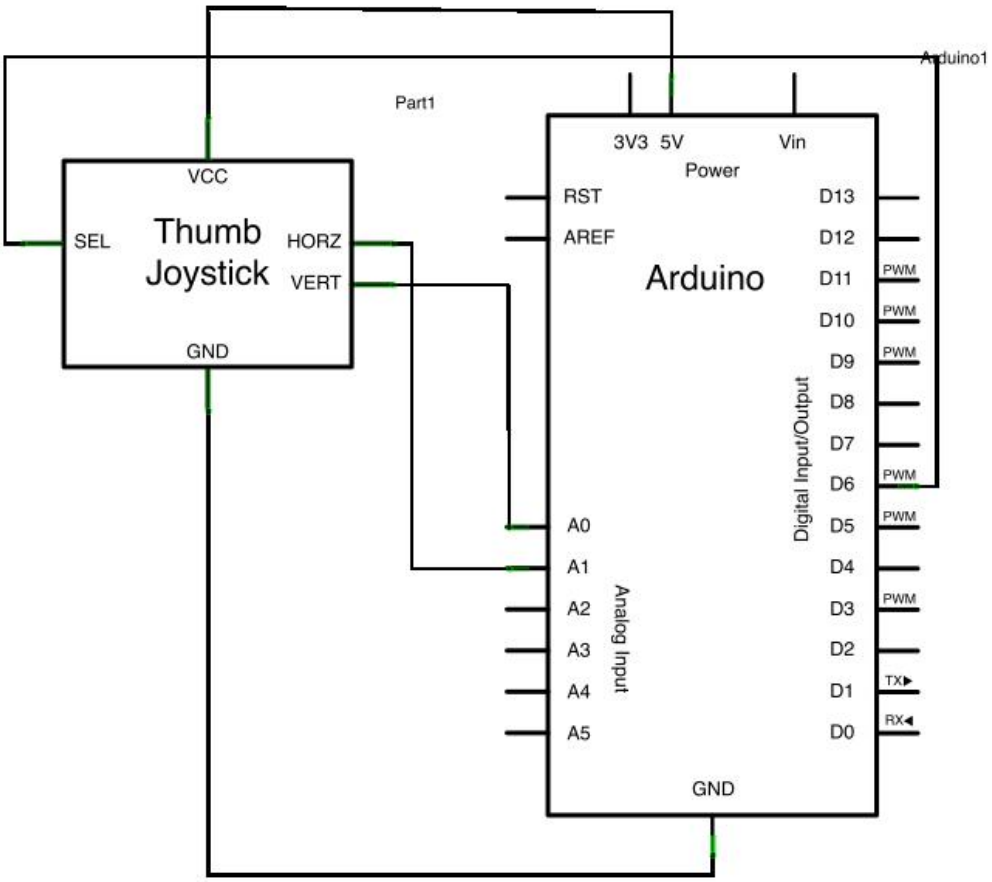
- Arduino Leonardo, Micro, or Due board
- 2 axis joystick
- momentary pushbutton (possibly integrated in the joystick)
- LED
- 220 ohm resistor
- 10k ohm resistor (if needed as pulldown)

### Circuit

Connect your Leonardo board to your computer with a micro-USB cable. The pushbutton is connected to pin 6. If you're using a part like the Joystick shield pictured below, you may not need a pulldown resistor. The x-axis on the joystick is connected to analog in 0, the y-axis is on analog in 1.




## Schematic




Code

```
/*
  JoystickMouseControl

Controls the mouse from a joystick on an Arduino Leonardo, Micro or Due.
Uses a pushbutton to turn on and off mouse control, and
a second pushbutton to click the left mouse button

Hardware:
* 2-axis joystick connected to pins A0 and A1
* pushbuttons connected to pin D2 and D3

The mouse movement is always relative. This sketch reads
two analog inputs that range from 0 to 1023 (or less on either end)
and translates them into ranges of -6 to 6.
The sketch assumes that the joystick resting values are around the
middle of the range, but that they vary within a threshold.

WARNING:  When you use the Mouse.move() command, the Arduino takes
over your mouse!  Make sure you have control before you use the command.
This sketch includes a pushbutton to toggle the mouse control state, so
```

```
 you can turn on and off mouse control.

 created 15 Sept 2011
 updated 28 Mar 2012
 by Tom Igoe

 this code is in the public domain

 */

#include "Mouse.h"

// set pin numbers for switch, joystick axes, and LED:
const int switchPin = 2;        // switch to turn on and off mouse control
const int mouseButton = 3;     // input pin for the mouse pushButton
const int xAxis = A0;           // joystick X axis
const int yAxis = A1;           // joystick Y axis
const int ledPin = 5;           // Mouse control LED

// parameters for reading the joystick:
int range = 12;                 // output range of X or Y movement
int responseDelay = 5;          // response delay of the mouse, in ms
int threshold = range / 4;     // resting threshold
int center = range / 2;         // resting position value

boolean mouseIsActive = false;     // whether or not to control the mouse
int lastSwitchState = LOW;          // previous switch state

void setup() {
 pinMode(switchPin, INPUT);        // the switch pin
 pinMode(ledPin, OUTPUT);          // the LED pin
 // take control of the mouse:
  Mouse.begin();
}

void loop() {
 // read the switch:
 int switchState = digitalRead(switchPin);
 // if it's changed and it's high, toggle the mouse state:
 if (switchState != lastSwitchState) {
    if (switchState == HIGH) {
       mouseIsActive = !mouseIsActive;
      // turn on LED to indicate mouse state:
      digitalWrite(ledPin, mouseIsActive);
```

```
    }
  }
  // save switch state for next comparison:
   lastSwitchState = switchState;

  // read and scale the two axes:
  int xReading = readAxis(A0);
  int yReading = readAxis(A1);

  // if the mouse control state is active, move the mouse:
  if (mouseIsActive) {
     Mouse.move(xReading, yReading, 0);
  }

  // read the mouse button and click or not click:
  // if the mouse button is pressed:
  if (digitalRead(mouseButton) == HIGH) {
    // if the mouse is not pressed, press it:
    if (!Mouse.isPressed(MOUSE_LEFT)) {
       Mouse.press(MOUSE_LEFT);
    }
  }
  // else the mouse button is not pressed:
  else {
    // if the mouse is pressed, release it:
    if (Mouse.isPressed(MOUSE_LEFT)) {
       Mouse.release(MOUSE_LEFT);
    }
  }

  delay(responseDelay);
}

/*
  reads an axis (0 or 1 for x or y) and scales the
 analog input range to a range from 0 to <range>
 */

int readAxis(int thisAxis) {
  // read the analog input:
  int reading = analogRead(thisAxis);

  // map the reading from the analog input range to the output range:
   reading = map(reading, 0, 1023, 0, range);
```

```
 // if the output reading is outside from the
 // rest position threshold,   use it:
 int distance = reading - center;

 if (abs(distance) < threshold) {
     distance = 0;
 }

 // return the distance for this axis:
 return distance;
}
```