

# Language Reference

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

## Structure

setup()

loop()

### 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

```
int buttonPin = 3;
```

V1.0

---

```
// setup initializes serial and the button pin
void setup()
{
  beginSerial(9600);
  pinMode(buttonPin, INPUT);
}

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

  delay(1000);
}
```

## Control Structures

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

**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.

[Reference Home](#)

## 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.

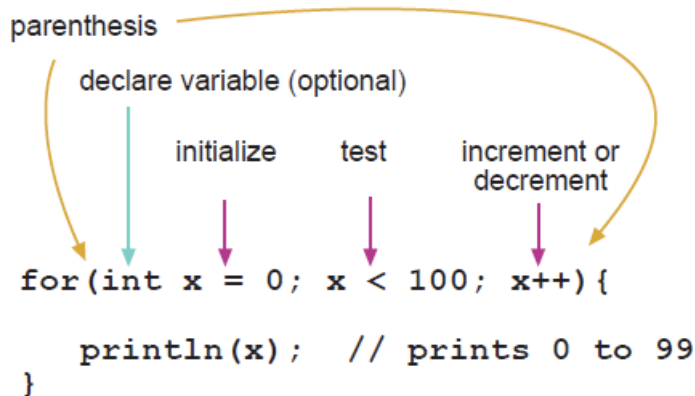
## for statements

### 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);
}
```



```
for(int x = 0; x < 100; x++){
  println(x); // prints 0 to 99
}
```

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 statements**

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

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

**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
```

V1.0

```
// default is optional
}
```

**Syntax**

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

**Parameters**

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

**label:** a value to compare the variable to

**while loops****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
{
```

V1.0

```

    // statement block
} while (test condition);

```

**Example**

```

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

} while (x < 100);

```

**break**

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

**Example**

```

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

```

**continue**

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

**Example**

```

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

    digitalWrite(PWMPin, x);
    delay(50);
}

```



## return

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

**Syntax:**

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

**Parameters**

value: any variable or constant type

**Examples:**

A function to compare a sensor input to a threshold

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

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

```
void loop() {  
  
    // brilliant code idea to test here  
  
    return;  
  
    // the rest of a dysfunctional sketch here  
    // this code will never be executed  
}
```

## goto

Transfers program flow to a labeled point in the program

**Syntax**

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

**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.

V1.0

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

### ; 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)  
}
```

V1.0

---

```
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
```

V1.0

```

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.

**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](#).

V1.0

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)

± (addition)

− (subtraction)

\* (multiplication)

/ (division)

% (modulo)

### = 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;
```



V1.0

```
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)

V1.0

`!=` (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.

[Reference Home](#)

## 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) {
```

V1.0

```
// ...
}
```

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
```

### See also

`&` (bitwise AND)

`|` (bitwise OR)

`~` (bitwise NOT)

## The pointer operators

### `&` (reference) and `*` (dereference)

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

## Bitwise Operators

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

### 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
```

V1.0

```
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

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 2,3 & 7 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 bitwise 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.



V1.0

```
// 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);
}
```

See Also

&&(Boolean AND)||(Boolean OR)[Reference Home](#)

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

### Description

From *The Bitmath Tutorial* 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 [here](#).

### Syntax

variable &lt;&lt; number\_of\_bits

variable &gt;&gt; number\_of\_bits

### Parameters

variable - (byte, int, long) number\_of\_bits integer &lt;= 32

### Example:

```
int a = 5;           // binary: 00000000000000101
int b = a << 3;      // binary: 0000000000101000, or 40 in decimal
int c = b >> 3;      // binary: 00000000000000101, 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: 00000000000000101
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

V1.0

raised to the right operand power. For example, to generate powers of 2, the following expressions can be employed:

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

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

```
int x = -16;      // binary: 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)  
`+=` (compound addition)  
`-=` (compound subtraction)  
`*=` (compound multiplication)  
`/=` (compound division)  
`&=` (compound bitwise and)  
`|=` (compound bitwise or)

### **++ (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
  
```

### **+= , -= , \*= , /=**

#### **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;
  
```

V1.0

**Parameters**

x: any variable type

y: any variable type or constant

**Examples**

```

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

```

**compound bitwise AND (&=)****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 = 10101010;
```

```
myByte &= B1111100 == B10101000;
```

See Also

`|=` (compound bitwise or)

`&` (bitwise AND)

`|` (bitwise OR)

## 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 ORed" with 0 are unchanged, so if myByte is a byte variable,  
`myByte | 00000000 = myByte;`

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

```
myByte | B1111111 = B1111111;
```

V1.0

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 = B10101010;`

`myByte |= B00000011 == B10101011;`

## Variables

## Constants

HIGH | LOW

INPUT | OUTPUT | INPUT\_PULLUP

true | false

integer constants

floating point constants

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

### Defining Logical Levels, true and false (Boolean Constants)

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

#### false

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

#### true

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

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

### Defining Pin Levels, HIGH and LOW

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

#### HIGH

The meaning of HIGH (in reference to a pin) is somewhat different depending on whether a pin is set to an INPUT or OUTPUT. When a pin is configured as an INPUT with pinMode, and read with digitalRead, the microcontroller will report HIGH if a voltage of 3 volts or more is present at the pin.

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

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

#### LOW

The meaning of LOW also has a different meaning depending on whether a pin is set to INPUT or OUTPUT. When a pin is configured as an INPUT with pinMode, and read with digitalRead, the microcontroller will report LOW if a voltage of 2 volts or less is present at the pin.

When a pin is configured to OUTPUT with pinMode, and set to LOW with digitalWrite, the pin is at 0 volts. In this state it can *sink* current, e.g. light an LED

that is connected through a series resistor to, +5 volts, or to another pin configured as an output, and set to HIGH.

#### Defining Digital Pins, 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, but not powering an LED.

If you have your pin configured as an INPUT, you will want the pin to have a reference to ground, often accomplished with a pull-down resistor (a resistor going to ground) as described in the [Digital Read Serial](#) tutorial.

##### Pins Configured as INPUT\_PULLUP

The Atmega chip 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-down resistors, you can use the **INPUT\_PULLUP** argument in `pinMode()`. This effectively inverts the behavior, where HIGH means the sensor is off, and LOW means the sensor is on. See the [Input Pullup Serial](#) tutorial for an example of this in use.

##### Pins Configured as Outputs

Pins configured as **OUTPUT** with `pinMode()` are said to be in a low-impedance state. This means that they can provide a substantial amount of current to other circuits. Atmega pins can source (provide positive current) or sink (provide negative current) up to 40 mA (milliamps) of current to other devices/circuits. This makes them useful for powering LED's but useless for reading sensors. Pins configured as outputs can also be damaged or destroyed if short circuited to either ground or 5 volt power rails. The amount of current provided by an Atmega pin is also not enough to power most relays or motors, and some interface circuitry will be required

## Data Types



V1.0

voidbooleancharunsigned charbyteintunsigned intwordlongunsigned longfloatdoublestring - char arrayString - objectarray

## 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()
```

V1.0

```

{
  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 equivalent

```

## 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.

V1.0

**Example**

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

## int

**Description**

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

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

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

**Example**

```
int ledPin = 13;
```

**Syntax**

```
int var = val;
```

var - your int variable name

val - the value you assign to that variable

**Coding Tip**

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

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

x = 32,767;
x = x + 1; // x now contains -32,768 - rolls over
```

## unsigned int

**Description**

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

**Example**

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

V1.0

**Syntax**

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

var - your unsigned int variable name

val - the value you assign to that variable

**Coding Tip**

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

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

**word****Description**

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

**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.

**Example**

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

**Syntax**

```
long var = val;
```

var - the long variable name

val - the value assigned to the variable

**unsigned long****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);
```

V1.0

}

```

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

**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.

**Examples**

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

**Syntax**

```
float var = val;
```

var - your float variable name

val - the value you assign to that variable

V1.0

**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. Occupies 4 bytes.

The double implementation on the Arduino is currently exactly the same as the float, with no gain in precision.

**Tip**

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

**string****Description**

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

**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);  
  }  
}
```

## 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.



V1.0

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

**To assign a value to an array:**

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

**To retrieve a value from an array:**

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

**Arrays and FOR Loops**

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

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

**Example**

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

## Conversion

[char\(\)](#)

[byte\(\)](#)

[int\(\)](#)

[word\(\)](#)

[long\(\)](#)

[float\(\)](#)

### char()

**Description**

Converts a value to the [char](#) data type.

**Syntax**

```
char(x)
```

**Parameters**

x: a value of any type

V1.0

**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

V1.0

**Returns**

long

**float()****Description**

Converts a value to the float data type.

**Syntax**`float(x)`**Parameters**

x: a value of any type

**Returns**`float`**Notes**

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

## Variable Scope & Qualifiers

variable scopestaticvolatileconst

### Variable Scope

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

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

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

V1.0

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;
```

V1.0

```

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 keyword

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

V1.0

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

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

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

### Example

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

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

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

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

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

## const keyword

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

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

V1.0

**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()**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;
```

V1.0

```
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)) - 1; 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.



V1.0

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>
```

[\[Get Code\]](#)

## Syntax

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

[\[Get Code\]](#)

- 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
```

[\[Get Code\]](#)

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.

V1.0

```
#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:
```

```
}
```

[\[Get Code\]](#)

V1.0

## 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);
```

V1.0

```
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 );
}
}
```

[\[Get Code\]](#)

## 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";
```

[\[Get Code\]](#)

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"
```

[\[Get Code\]](#)

## The F() macro

When an instruction like :

V1.0

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

[\[Get Code\]](#)

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\(\)](#)  
[digitalWrite\(\)](#)  
[digitalRead\(\)](#)

### 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  
}
```

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

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

V1.0

```
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**

```
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
}
```

V1.0

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

### 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()`

`analogRead()`

`analogWrite()` - PWM

## `analogReference(type)`

### 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.

### 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.



V1.0

**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  $2.5 * 32 / (32 + 5) = \sim 2.2V$  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
```

V1.0

```
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 is approximately 490 Hz.

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

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

### Syntax

**analogWrite(pin, value)**

### Parameters

pin: the pin to write to.

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

### Returns

nothing

### Notes and Known Issues

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

### Example

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

V1.0

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

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

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

Advanced I/O

tone()  
noTone()  
shiftOut()  
shiftIn()  
pulseIn()

## 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.

V1.0

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).

**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)`

V1.0

**Parameters****dataPin**: the pin on which to output each bit (*int*)**clockPin**: the pin to toggle once the **dataPin** has been set to the correct value (*int*)**bitOrder**: which order to shift out the bits; either **MSBFIRST** or **LSBFIRST**.

(Most Significant Bit First, or, Least Significant Bit First)

**value**: the data to shift out. (*byte*)**Returns**

None

**Note**

The **dataPin** and **clockPin** must already be configured as outputs by a call to [pinMode\(\)](#).

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

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

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

[\[Get Code\]](#)**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;
```

V1.0

```

//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.

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)

V1.0

**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. Gives up and returns 0 if no pulse starts within a specified time out.

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

**Syntax**`pulseIn(pin, value)``pulseIn(pin, value, timeout)`**Parameters**

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

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

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

**Returns**

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

**Example**

```
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*)

**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);
}
```

**Tip:**

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



## micros()

### 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 milliseconds) specified as parameter. (There are 1000 milliseconds in a second.)

### Syntax

delay(ms)

### Parameters

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

### Returns

nothing

### Example

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

V1.0

```
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 with 100 microseconds period.

#### **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\(\)](#)  
[max\(\)](#)  
[abs\(\)](#)  
[constrain\(\)](#)  
[map\(\)](#)  
[pow\(\)](#)  
[sqrt\(\)](#)

## 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
```

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

## max(x, y)

**Description**

Calculates the maximum of two numbers.

**Parameters**

x: the first number, any data type

y: the second number, any data type

**Returns**

The larger of the two parameter values.

**Example**

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

**Note**

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

V1.0

**Warning**

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

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

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

**abs(x)****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
```

```
a++; // use this instead -  
abs(a); // 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()cos()tan()**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*)

## Random Numbers

randomSeed()



random()**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 (*optional*)

max - upper bound of the random value, exclusive

V1.0

**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()highByte()bitRead()bitWrite()bitSet()bitClear()bit()**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

V1.0

**Returns**

the value of the bit

## External Interrupts

[attachInterrupt\(\)](#)[detachInterrupt\(\)](#)**attachInterrupt()****Description**

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

**Syntax**

`attachInterrupt(interrupt, function, mode)`

**Parameters**

**interrupt:** the number of the interrupt (*int*)

**function:** the function to call when the interrupt occurs; this function must take no parameters and return nothing. This function is sometimes referred to as an *interrupt service routine*.

**mode** defines when the interrupt should be triggered. Four constants are predefined as valid values:

**LOW** to trigger the interrupt whenever the pin is low,

**CHANGE** to trigger the interrupt whenever the pin changes value

**RISING** to trigger when the pin goes from low to high,

**FALLING** for when the pin goes from high to low.

**Returns**

none

**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*

V1.0

*should declare as volatile any variables that you modify within the attached function.*

### Using Interrupts

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

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

### Example

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

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

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

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

## detachInterrupt()

### Description

Turns off the given interrupt.

### Syntax

```
detachInterrupt(interrupt)
```

### Parameters

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

### See also

## Interrupts

interrupts()  
noInterrupts()

**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.

V1.0

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**  
**Stream****Serial**

Used for communication between the Arduino board and a computer or other devices. All Arduino boards have at least one serial port (also known as a UART or USART): **Serial**. It communicates on digital pins 0 (RX) and 1 (TX) as well as with the computer via USB. Thus, if you use these functions, you cannot also use pins 0 and 1 for digital input or output.



V1.0

You can use the Arduino environment's built-in serial monitor to communicate with an Arduino board. Click the serial monitor button in the toolbar and select the same baud rate used in the call to `begin()`.

The [Arduino Mega](#) has three additional serial ports: **Serial1** on pins 19 (RX) and 18 (TX), **Serial2** on pins 17 (RX) and 16 (TX), **Serial3** on pins 15 (RX) and 14 (TX). To use these pins to communicate with your personal computer, you will need an additional USB-to-serial adaptor, as they are not connected to the Mega's USB-to-serial adaptor. To use them to communicate with an external TTL serial device, connect the TX pin to your device's RX pin, the RX to your device's TX pin, and the ground of your Mega to your device's ground. (Don't connect these pins directly to an RS232 serial port; they operate at +/- 12V and can damage your Arduino board.)

The Arduino Leonardo board uses **Serial1** to communicate via RS232 on pins 0 (RX) and 1 (TX). **Serial** is reserved for USB CDC communication. For more information, refer to the [Leonardo getting started](#) page and [hardware page](#).

### Functions

[`if \(Serial\)`](#)  
[`available\(\)`](#)  
[`begin\(\)`](#)  
[`end\(\)`](#)  
[`find\(\)`](#)  
[`findUntil\(\)`](#)  
[`flush\(\)`](#)  
[`parseFloat\(\)`](#)  
[`parseInt\(\)`](#)  
[`peek\(\)`](#)  
[`print\(\)`](#)  
[`println\(\)`](#)  
[`read\(\)`](#)  
[`readBytes\(\)`](#)  
[`readBytesUntil\(\)`](#)  
[`setTimeout\(\)`](#)  
[`write\(\)`](#)  
[`serialEvent\(\)`](#)

### Examples

[ReadASCIIString](#)  
[ASCII Table](#)  
[Dimmer](#)  
[Graph](#)  
[Physical Pixel](#)  
[Virtual Color Mixer](#)  
[Serial Call Response](#)  
[Serial Call Response ASCII](#)

## 1 if (Serial)

### Description

Indicates if the specified Serial port is ready.

On the Leonardo, **if (Serial)** indicates whether or not the USB CDC serial connection is open. For all other instances, including **if (Serial1)** on the Leonardo, this will always return true.

This was introduced in Arduino 1.0.1.

### Syntax

*All boards:*

`if (Serial)`

*Arduino Leonardo specific:*

`if (Serial1)`

*Arduino Mega specific:*

`if (Serial1)`

`if (Serial2)`

`if (Serial3)`

### Parameters

none

### Returns

boolean : returns true if the specified serial port is available. This will only return false if querying the Leonardo's USB CDC serial connection before it is ready.

### Example:

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

void loop() {
  //proceed normally
}
```

## 2 available()

### Description

Get the number of bytes (characters) available for reading from the serial port. This is data that's already arrived and stored in the serial receive buffer (which holds 64 bytes). `available()` inherits from the Stream utility class.

V1.0

**Syntax**`Serial.available()`*Arduino Mega only:*`Serial1.available()``Serial2.available()``Serial3.available()`**Parameters**

none

**Returns**

the number of bytes available to read

**Example**

```
int incomingByte = 0;    // for incoming serial data

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

void loop() {

    // send data only when you receive data:
    if (Serial.available() > 0) {
        // read the incoming byte:
        incomingByte = Serial.read();

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

[\[Get Code\]](#)**Arduino Mega example:**

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

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

V1.0

```

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

```

### 3 begin()

#### Description

Sets the data rate in bits per second (baud) for serial data transmission. For communicating with the computer, use one of these rates: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200. You can, however, specify other rates - for example, to communicate over pins 0 and 1 with a component that requires a particular baud rate.

#### Syntax

`Serial.begin(speed)`

*Arduino Mega only:*

`Serial1.begin(speed)`

`Serial2.begin(speed)`

`Serial3.begin(speed)`

#### Parameters

speed: in bits per second (baud) - *long*

#### Returns

nothing

#### Example:

```

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

```

```

void loop() {}

```

[\[Get Code\]](#)

#### Arduino Mega example:

```

// Arduino Mega using all four of its Serial ports
// (Serial, Serial1, Serial2, Serial3),
// with different baud rates:

```

```

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

```

V1.0

```

Serial.println("Hello Computer");
Serial1.println("Hello Serial 1");
Serial2.println("Hello Serial 2");
Serial3.println("Hello Serial 3");
}

```

```
void loop() {}
```

[\[Get Code\]](#)*Thanks to Jeff Gray for the mega example***See also**

## 4 end()

**Description**

Disables serial communication, allowing the RX and TX pins to be used for general input and output. To re-enable serial communication, call [Serial.begin\(\)](#).

**Syntax**

```
Serial.end()
```

*Arduino Mega only:*

```
Serial1.end()
```

```
Serial2.end()
```

```
Serial3.end()
```

**Parameters**

none

**Returns**

nothing

## 5 Serial.find()

**Description**

`Serial.find()` reads data from the serial buffer until the target string of given length is found. The function returns true if target string is found, false if it times out.

`Serial.flush()` inherits from the [Stream](#) utility class.

**Syntax**

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

**Parameters**

*target* : the string to search for (char)

**Returns**

boolean

**See als**

## 6 Serial.findUntil()

**Description**

`Serial.findUntil()` reads data from the serial buffer until a target string of given length or terminator string is found.

The function returns true if the target string is found, false if it times out.

V1.0

`Serial.findUntil()` inherits from the Stream utility class.

**Syntax**

`Serial.findUntil(target, terminal)`

**Parameters**

*target* : the string to search for (char)

*terminal* : the terminal string in the search (char)

**Returns**

boolean

**See als**

## 7 flush()

**Description**

Waits for the transmission of outgoing serial data to complete. (Prior to Arduino 1.0, this instead removed any buffered incoming serial data.)

`flush()` inherits from the Stream utility class.

**Syntax**

`Serial.flush()`

*Arduino Mega only:*

`Serial1.flush()`

`Serial2.flush()`

`Serial3.flush()`

**Parameters**

none

**Returns**

nothing

**See als**

## 8 Serial.parseFloat()

**Description**

`Serial.parseFloat()` returns the first valid floating point number from the Serial buffer.

Characters that are not digits (or the minus sign) are skipped. `parseFloat()` is terminated by the first character that is not a floating point number.

`Serial.parseFloat()` inherits from the Stream utility class.

**Syntax**

`Serial.parseFloat()`

**Parameters**

none

**Returns**

float

## 9 parseInt()

**Description**

Looks for the next valid integer in the incoming serial stream. `parseInt()` inherits from the Stream utility class.

V1.0

**Syntax**`Serial.parseInt()`*Arduino Mega only:*`Serial1.parseInt()``Serial2.parseInt()``Serial3.parseInt()`**Parameters**

none

**Returns**

int : the next valid integer

**Example****10 peek()****Description**

Returns the next byte (character) of incoming serial data without removing it from the internal serial buffer. That is, successive calls to `peek()` will return the same character, as will the next call to `read()`. `peek()` inherits from the Stream utility class.

**Syntax**`Serial.peek()`*Arduino Mega only:*`Serial1.peek()``Serial2.peek()``Serial3.peek()`**Parameters**

None

**Returns**

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

**11 print()****Description**

Prints data to the serial port as human-readable ASCII text. This command can take many forms. Numbers are printed using an ASCII character for each digit. Floats are similarly printed as ASCII digits, defaulting to two decimal places. Bytes are sent as a single character. Characters and strings are sent as is. For example:

`Serial.print(78)` gives "78"`Serial.print(1.23456)` gives "1.23"`Serial.print('N')` gives "N"

V1.0

`Serial.print("Hello world.")` gives "Hello world."

An optional second parameter specifies the base (format) to use; permitted values are BIN (binary, or base 2), OCT (octal, or base 8), DEC (decimal, or base 10), HEX (hexadecimal, or base 16). For floating point numbers, this parameter specifies the number of decimal places to use. For example:

`Serial.print(78, BIN)` gives "1001110"

`Serial.print(78, OCT)` gives "116"

`Serial.print(78, DEC)` gives "78"

`Serial.print(78, HEX)` gives "4E"

`Serial.println(1.23456, 0)` gives "1"

`Serial.println(1.23456, 2)` gives "1.23"

`Serial.println(1.23456, 4)` gives "1.2346"

You can pass flash-memory based strings to `Serial.print()` by wrapping them with `F()`. For example :

`Serial.print(F("Hello World"))`

To send a single byte, use `Serial.write()`.

### Syntax

`Serial.print(val)`

`Serial.print(val, format)`

### Parameters

val: the value to print - any data type

format: specifies the number base (for integral data types) or number of decimal places (for floating point types)

### Returns

size\_t (long): `print()` returns the number of bytes written, though reading that number is optional

### Example:

```
/*
  Uses a FOR loop for data and prints a number in various formats.
  */
int x = 0;      // variable

void setup() {
  Serial.begin(9600);      // open the serial port at 9600 bps:
}

void loop() {
  // print labels
  Serial.print("NO FORMAT");      // prints a label
  Serial.print("\t");      // prints a tab

  Serial.print("DEC");
  Serial.print("\t");
```



V1.0

```

Serial.print("HEX");
Serial.print("\t");

Serial.print("OCT");
Serial.print("\t");

Serial.print("BIN");
Serial.print("\t");

for(x=0; x< 64; x++){    // only part of the ASCII chart, change
to suit

    // print it out in many formats:
    Serial.print(x);      // print as an ASCII-encoded decimal -
same as "DEC"
    Serial.print("\t");   // prints a tab

    Serial.print(x, DEC); // print as an ASCII-encoded decimal
    Serial.print("\t");   // prints a tab

    Serial.print(x, HEX); // print as an ASCII-encoded hexadecimal
    Serial.print("\t");   // prints a tab

    Serial.print(x, OCT); // print as an ASCII-encoded octal
    Serial.print("\t");   // prints a tab

    Serial.println(x, BIN); // print as an ASCII-encoded binary
    //                        then adds the carriage return with
"println"
    delay(200);             // delay 200 milliseconds
}
Serial.println("");        // prints another carriage return
}

```

[\[Get Code\]](#)**Programming Tips**

As of version 1.0, serial transmission is asynchronous; `Serial.print()` will return before any characters are transmitted.

**See also****12 println()****Description**

Prints data to the serial port as human-readable ASCII text followed by a carriage return character (ASCII 13, or '\r') and a newline character (ASCII 10, or '\n'). This command takes the same forms as [Serial.print\(\)](#).

V1.0

**Syntax**`Serial.println(val)``Serial.println(val, format)`**Parameters**`val`: the value to print - any data type`format`: specifies the number base (for integral data types) or number of decimal places (for floating point types)**Returns**`size_t` (long): `println()` returns the number of bytes written, though reading that number is optional**Example:**

```
/*
  Analog input

  reads an analog input on analog in 0, prints the value out.

  created 24 March 2006
  by Tom Igoe
  */

int analogValue = 0;    // variable to hold the analog value

void setup() {
  // open the serial port at 9600 bps:
  Serial.begin(9600);
}

void loop() {
  // read the analog input on pin 0:
  analogValue = analogRead(0);

  // print it out in many formats:
  Serial.println(analogValue);    // print as an ASCII-encoded
  decimal
  Serial.println(analogValue, DEC); // print as an ASCII-encoded
  decimal
  Serial.println(analogValue, HEX); // print as an ASCII-encoded
  hexadecimal
  Serial.println(analogValue, OCT); // print as an ASCII-encoded
  octal
  Serial.println(analogValue, BIN); // print as an ASCII-encoded
  binary
}
```

V1.0

```
// delay 10 milliseconds before the next reading:
delay(10);
}
```

[\[Get Code\]](#)**See also****13 read()****Description**

Reads incoming serial data. `read()` inherits from the [Stream](#) utility class.

**Syntax**

`Serial.read()`

*Arduino Mega only:*

`Serial1.read()`

`Serial2.read()`

`Serial3.read()`

**Parameters**

None

**Returns**

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

**Example**

```
int incomingByte = 0; // for incoming serial data

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

void loop() {
    // send data only when you receive data:
    if (Serial.available() > 0) {
        // read the incoming byte:
        incomingByte = Serial.read();

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

[

## 14 Serial.readBytes()

### Description

Serial.readBytes() reads characters from the serial port into a buffer. The function terminates if the determined length has been read, or it times out (see [Serial.setTimeout\(\)](#)).

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

Serial.readBytes() inherits from the [Stream](#) utility class.

### Syntax

Serial.readBytes(buffer, length)

### Parameters

buffer: the buffer to store the bytes in (char[] or byte[])

length : the number of bytes to read (int)

### Returns

byte

### See

## 15 Serial.readBytesUntil()

### Description

Serial.readBytesUntil() reads characters from the serial buffer into an array. The function terminates if the terminator character is detected, the determined length has been read, or it times out (see [Serial.setTimeout\(\)](#)).

Serial.readBytesUntil() returns the number of characters read into the buffer. A 0 means no valid data was found.

Serial.readBytesUntil() inherits from the [Stream](#) utility class.

### Syntax

Serial.readBytesUntil(*character*, *buffer*, *length*)

### Parameters

*character* : the character to search for (char)

*buffer*: the buffer to store the bytes in (char[] or byte[]) *length* : the number of bytes to read (int)

### Returns

byte

### See

## 16 Serial.setTimeout()

### Description

Serial.setTimeout() sets the maximum milliseconds to wait for serial data when using [Serial.readBytesUntil\(\)](#) or [Serial.readBytes\(\)](#). It defaults to 1000 milliseconds.

Serial.setTimeout() inherits from the [Stream](#) utility class.

### Syntax

Serial.setTimeout(*time*)

V1.0

**Parameters***time* : timeout duration in milliseconds (long).**Parameters**

None

**See a**

## 17 write()

**Description**

Writes binary data to the serial port. This data is sent as a byte or series of bytes; to send the characters representing the digits of a number use the `print()` function instead.

**Syntax**`Serial.write(val)``Serial.write(str)``Serial.write(buf, len)`

*Arduino Mega also supports: Serial1, Serial2, Serial3 (in place of Serial)*

**Parameters***val*: a value to send as a single byte*str*: a string to send as a series of bytes*buf*: an array to send as a series of bytes*len*: the length of the buffer**Returns**

byte

`write()` will return the number of bytes written, though reading that number is optional

**Example**

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

```
void loop() {
    Serial.write(45); // send a byte with the value 45
```

```
    int bytesSent = Serial.write("hello"); //send the string "hello"
    and return the length of the string.
}
```

## 18 serialEvent()

**Description**

Called when data is available. Use `Serial.read()` to capture this data.

**Syntax**

```
void serialEvent() {
```

V1.0

```
//statements
}
```

[\[Get Code\]](#)*Arduino Mega only:*

```
void serialEvent1() {
  //statements
}
```

```
void serialEvent2() {
  //statements
}
```

```
void serialEvent3() {
  //statements
}
```

[\[Get Code\]](#)**Parameters**

statements: any valid statements

**Examples**[ReadASCIIString](#)[ASCII Table](#)[Dimmer](#)[Graph](#)[Physical Pixel](#)[Virtual Color Mixer](#)[Serial Call Response](#)[Serial Call Response ASCII](#)**Read ASCII String**

This sketch uses the `Serial.parseInt()` function to locate values separated by a non-alphanumeric character. Often people use a comma to indicate different pieces of information (this format is commonly referred to as **comma-separated-values**), but other characters like a space or a period will work too. The values are parsed into ints and used to determine the color of a RGB LED. You'll use the serial monitor to send strings like "5,220,70" to the Arduino to change the lights.

**Hardware Required**

Arduino Board

Breadboard

Hookup wire

Common anode RGB LED

Three 220-ohm resistors

V1.0

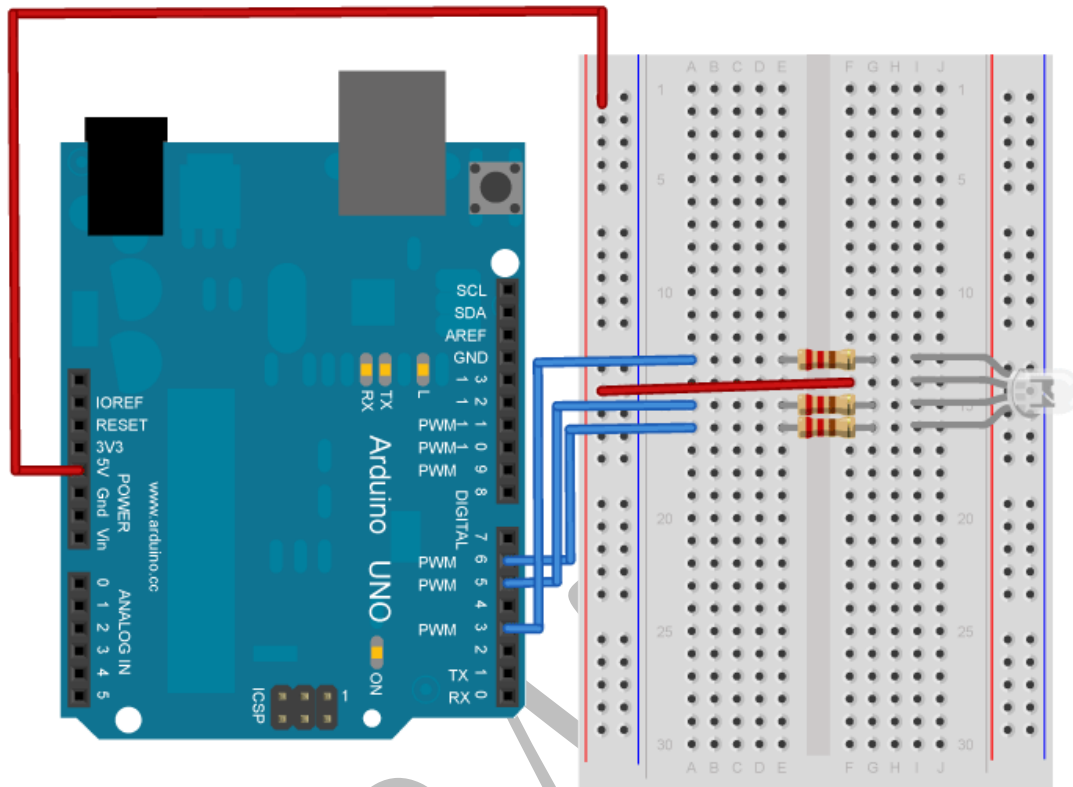
**Circuit**

image developed using [Fritzing](#). For more circuit examples, see the [Fritzing project page](#). You'll need five wires to make the circuit above. Connect a red wire to one of the long vertical rows on your breadboard. Connect the other end to the 5V pin on your Arduino.

Place an RGB LED on your breadboard. Check the datasheet for your specific LED to verify the pins. Connect the power rail you just created to the common anode on the LED.

With your remaining wires, connect your red cathode to pin 3, green cathode to pin 5, and blue cathode to pin 6 in series with the resistors.

RGB LEDs with a common anode share a common power pin. Instead of turning a pin HIGH to illuminate the LED, you need to turn the pin LOW, to create a voltage difference across the diode. So sending 255 via `analogWrite()` turns the LED off, while a value of 0 turns it on at full brightness. In the code below, you'll use a little bit of math on the Arduino side, so you can send values which correspond to the expected brightness. Essentially, instead of using `analogWrite(pin, brightness)`, you'll be calling `analogWrite(pin, 255-brightness)`.

V1.0

**Code**

You'll first set up some global variables for the pins your LED will connect to. This will make it easier to differentiate which one is red, green, and blue in the main part of your program:

```
const int redPin = 3;
const int greenPin = 5;
const int bluePin = 6;
```

In your setup(), begin serial communication at 9600 bits of data per second between Arduino and your computer with the line:

```
Serial.begin(9600);
```

Also in the setup, you'll want to configure the pins as outputs:

```
pinMode(redPin, OUTPUT);\ \ pinMode(greenPin, OUTPUT);
pinMode(bluePin, OUTPUT);
```

In the loop(), check to see if there is any data in the serial buffer. By making this a while() statement, it will run as long as there is information waiting to be read :

```
while (Serial.available() > 0) {
```

Next, declare some local variables for storing the serial information. This will be the brightness of the LEDs.

Using Serial.parseInt() to separate the data by commas, read the information into your variables:

```
int red = Serial.parseInt();\ \ int green = Serial.parseInt();
int blue = Serial.parseInt();
```

Once you've read the data into your variables, check for the newline character to proceed:

```
if (Serial.read() == '\n') {
```

Using constrain(), you can keep the values in an acceptable range for PWM control. This way, if the value was outside the range of what PWM can send, it will be limited to a valid number. By subtracting this value from 255 you will be formatting the value to use with a common anode LED. As explained above, these LEDs will illuminate when there is a voltage difference between the anode and the pin connected to the Arduino:

```
red = 255 - constrain(red, 0, 255);
green = 255 - constrain(green, 0, 255);
blue = 255 - constrain(blue, 0, 255);
```

Now that you have formatted the values for PWM, use analogWrite() to change the color of the LED. Because you subtracted your value from 255 in the step above:

```
analogWrite(redPin, red);
analogWrite(greenPin, green);
analogWrite(bluePin, blue);
```

Send the value of each LED back to the serial monitor in one string as HEX values :



V1.0

```

Serial.print(red, HEX);
Serial.print(green, HEX);
Serial.println(blue, HEX);
Finally, close up your brackets from the if statement, while
statement, and main loop :
}
}
}

```

Once you have programmed the Arduino, open your Serial minitor. Make sure you have chosen to send a newline character when sending a message. Enter values between 0-255 for the lights in the following format : Red,Green,Blue. Once you have sent the values to the Arduino, the attached LED will turn the color you specified, and you will receive the HEX values in the serial monitor.

```

/*
  Reading a serial ASCII-encoded string.

  This sketch demonstrates the Serial parseInt() function.
  It looks for an ASCII string of comma-separated values.
  It parses them into ints, and uses those to fade an RGB LED.

  Circuit: Common-anode RGB LED wired like so:
  * Red cathode: digital pin 3
  * Green cathode: digital pin 5
  * blue cathode: digital pin 6
  * anode: +5V

  created 13 Apr 2012
  by Tom Igoe

  This example code is in the public domain.
  */

// pins for the LEDs:
const int redPin = 3;
const int greenPin = 5;
const int bluePin = 6;

void setup() {
  // initialize serial:
  Serial.begin(9600);
  // make the pins outputs:
  pinMode(redPin, OUTPUT);
  pinMode(greenPin, OUTPUT);

```

V1.0

```

    pinMode(bluePin, OUTPUT);

}

void loop() {
    // if there's any serial available, read it:
    while (Serial.available() > 0) {

        // look for the next valid integer in the incoming serial stream:
        int red = Serial.parseInt();
        // do it again:
        int green = Serial.parseInt();
        // do it again:
        int blue = Serial.parseInt();

        // look for the newline. That's the end of your
        // sentence:
        if (Serial.read() == '\n') {
            // constrain the values to 0 - 255 and invert
            // if you're using a common-cathode LED, just use
            "constrain(color, 0, 255);"
            red = 255 - constrain(red, 0, 255);
            green = 255 - constrain(green, 0, 255);
            blue = 255 - constrain(blue, 0, 255);

            // fade the red, green, and blue legs of the LED:
            analogWrite(redPin, red);
            analogWrite(greenPin, green);
            analogWrite(bluePin, blue);

            // print the three numbers in one string as hexadecimal:
            Serial.print(red, HEX);
            Serial.print(green, HEX);
            Serial.println(blue, HEX);
        }
    }
}

```

### ASCII Table

Demonstrates the advanced serial printing functions by generating a table of characters and their ASCII values in decimal, hexadecimal, octal, and binary. For more on ASCII, see [asciitable.com](http://asciitable.com)

### Hardware Required

Arduino Board

V1.0

## Circuit

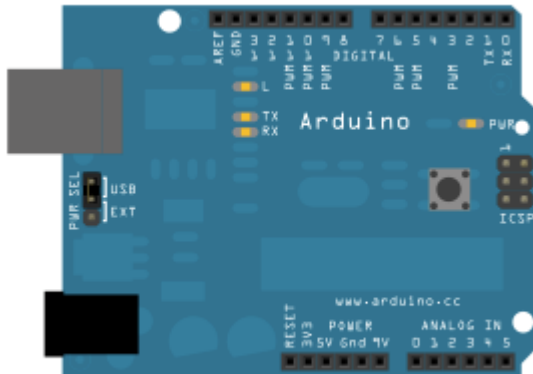


image developed using [Fritzing](#). For more circuit examples, see the [Fritzing project page](#). None, but the Arduino has to be connected to the computer.

## Code

```

/*
  ASCII table

  Prints out byte values in all possible formats:
  * as raw binary values
  * as ASCII-encoded decimal, hex, octal, and binary values

  For more on ASCII, see http://www.asciitable.com and
http://en.wikipedia.org/wiki/ASCII

  The circuit: No external hardware needed.

  created 2006
  by Nicholas Zambetti
  modified 9 Apr 2012
  by Tom Igoe

  This example code is in the public domain.

  <http://www.zambetti.com>

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

```

```
// prints title with ending line break
Serial.println("ASCII Table ~ Character Map");
}

// first visible ASCII character '!' is number 33:
int thisByte = 33;
// you can also write ASCII characters in single quotes.
// for example. '!' is the same as 33, so you could also use this:
//int thisByte = '!';

void loop() {
  // prints value unaltered, i.e. the raw binary version of the
  // byte. The serial monitor interprets all bytes as
  // ASCII, so 33, the first number, will show up as '!'
  Serial.write(thisByte);

  Serial.print(", dec: ");
  // prints value as string as an ASCII-encoded decimal (base 10).
  // Decimal is the default format for Serial.print() and
  Serial.println(),
  // so no modifier is needed:
  Serial.print(thisByte);
  // But you can declare the modifier for decimal if you want to.
  //this also works if you uncomment it:

  // Serial.print(thisByte, DEC);

  Serial.print(", hex: ");
  // prints value as string in hexadecimal (base 16):
  Serial.print(thisByte, HEX);

  Serial.print(", oct: ");
  // prints value as string in octal (base 8);
  Serial.print(thisByte, OCT);

  Serial.print(", bin: ");
  // prints value as string in binary (base 2)
  // also prints ending line break:
  Serial.println(thisByte, BIN);

  // if printed last visible character '~' or 126, stop:
  if(thisByte == 126) {      // you could also use if (thisByte ==
```

V1.0

```
'~') {
    // This loop loops forever and does nothing
    while(true) {
        continue;
    }
}
// go on to the next character
thisByte++;
}
```

[\[Get Code\]](#)**Output**

ASCII Table ~ Character Map

```
!, dec: 33, hex: 21, oct: 41, bin
4, dec: 34, hex: 22, oct: 42, bin: 100001
", dec: 35, hex: 23, oct: 43, bin: 100010
#, dec: 36, hex: 24, oct: 44, bin: 100011
$, dec: 37, hex: 25, oct: 45, bin: 100100
%, dec: 38, hex: 26, oct: 46, bin: 100101
&, dec: 39, hex: 27, oct: 47, bin: 100110
', dec: 40, hex: 28, oct: 50, bin: 100111
(, dec: 41, hex: 29, oct: 51, bin: 101000
), dec: 42, hex: 2A, oct: 52, bin: 101001
*, dec: 43, hex: 2B, oct: 53, bin: 101010
+, dec: 44, hex: 2C, oct: 54, bin: 101011
,, dec: 45, hex: 2D, oct: 55, bin: 101100
-, dec: 46, hex: 2E, oct: 56, bin: 101101
./, dec: 47, hex: 2F, oct: 57, bin: 101110
0, dec: 48, hex: 30, oct: 60, bin: 110000
1, dec: 49, hex: 31, oct: 61, bin: 110001
2, dec: 50, hex: 32, oct: 62, bin: 110010
3, dec: 51, hex: 33, oct: 63, bin: 110011
4, dec: 52, hex: 34, oct: 64, bin: 110100
5, dec: 53, hex: 35, oct: 65, bin: 110101
6, dec: 54, hex: 36, oct: 66, bin: 110110
7, dec: 55, hex: 37, oct: 67, bin: 110111
8, dec: 56, hex: 38, oct: 70, bin: 111000
9, dec: 57, hex: 39, oct: 71, bin: 111001
:, dec: 58, hex: 3A, oct: 72, bin: 111010
;, dec: 59, hex: 3B, oct: 73, bin: 111011
<, dec: 60, hex: 3C, oct: 74, bin: 111100
=, dec: 61, hex: 3D, oct: 75, bin: 111101
>, dec: 62, hex: 3E, oct: 76, bin: 111110
```

V1.0

?, dec: 63, hex: 3F, oct: 77, bin: 111111  
 @, dec: 64, hex: 40, oct: 100, bin: 1000000  
 A, dec: 65, hex: 41, oct: 101, bin: 1000001  
 B, dec: 66, hex: 42, oct: 102, bin: 1000010  
 C, dec: 67, hex: 43, oct: 103, bin: 1000011  
 D, dec: 68, hex: 44, oct: 104, bin: 1000100  
 E, dec: 69, hex: 45, oct: 105, bin: 1000101

## Dimmer

This example shows how to send data from a personal computer to an Arduino board to control the brightness of an LED. The data is sent in individual bytes, each of which ranges in value from 0 to 255. Arduino reads these bytes and uses them to set the brightness of the LED.

You can send bytes to the Arduino from any software that can access the computer serial port. Examples for [Processing](#) and [Max/MSP version 5](#) are shown below.

### Hardware Required

Arduino Board

LED

220 ohm resistor

### Software Required

[Processing](#) or

[Max/MSP version 5](#)

### Circuit

An LED connected to pin 9. Use an appropriate resistor as needed. For most common LEDs, you can usually do without the resistor, as the current output of the digital I/O pins is limited.

[click the image to enlarge](#)

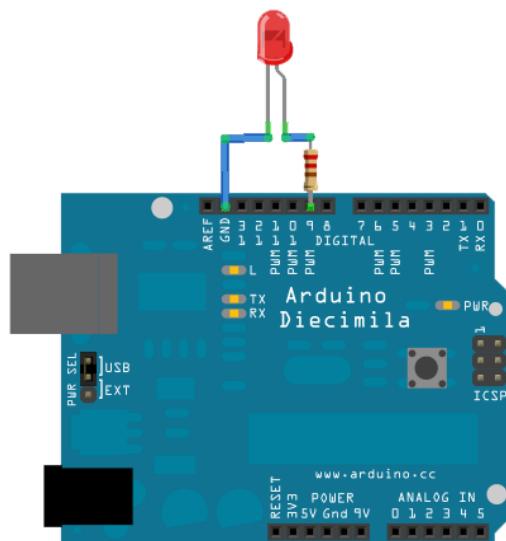
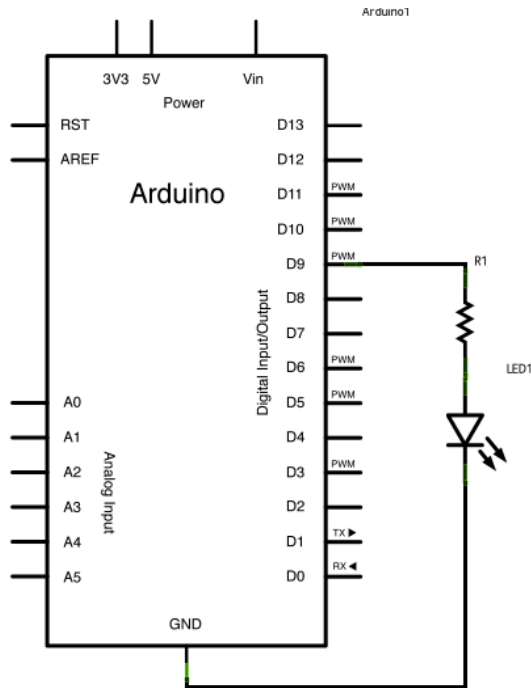


image developed using [Fritzing](#). For more circuit examples, see the [Fritzing project page](#)

V1.0

## Schematic

click the image to enlarge



%

## Code

/\*

*Dimmer*

*Demonstrates the sending data from the computer to the Arduino board,*

*in this case to control the brightness of an LED. The data is sent*

*in individual bytes, each of which ranges from 0 to 255. Arduino reads these bytes and uses them to set the brightness of the LED.*

*The circuit:*

*LED attached from digital pin 9 to ground.*

*Serial connection to Processing, Max/MSP, or another serial application*

*created 2006*

*by David A. Mellis*

*modified 30 Aug 2011*

*by Tom Igoe and Scott Fitzgerald*

*This example code is in the public domain.*

*<http://www.arduino.cc/en/Tutorial/Dimmer>*

V1.0

```

*/

const int ledPin = 9;      // the pin that the LED is attached to

void setup()
{
    // initialize the serial communication:
    Serial.begin(9600);
    // initialize the ledPin as an output:
    pinMode(ledPin, OUTPUT);
}

void loop() {
    byte brightness;

    // check if data has been sent from the computer:
    if (Serial.available()) {
        // read the most recent byte (which will be from 0 to 255):
        brightness = Serial.read();
        // set the brightness of the LED:
        analogWrite(ledPin, brightness);
    }
}

/* Processing code for this example
// Dimmer - sends bytes over a serial port
// by David A. Mellis
//This example code is in the public domain.

import processing.serial.*;
Serial port;

void setup() {
    size(256, 150);

    println("Available serial ports:");
    println(Serial.list());

    // Uses the first port in this list (number 0). Change this to
    // select the port corresponding to your Arduino board. The last
    // parameter (e.g. 9600) is the speed of the communication. It
    // has to correspond to the value passed to Serial.begin() in your
    // Arduino sketch.
    port = new Serial(this, Serial.list()[0], 9600);

```



```

// If you know the name of the port used by the Arduino board, you
// can specify it directly like this.
//port = new Serial(this, "COM1", 9600);
}

void draw() {
// draw a gradient from black to white
for (int i = 0; i < 256; i++) {
stroke(i);
line(i, 0, i, 150);
}

// write the current X-position of the mouse to the serial port
as
// a single byte
port.write(mouseX);
}
*/

/* Max/MSP v5 patch for this example

-----begin_max5_patcher-----
1008.3ocuXszaiaCD9r8uhA5rqAeHla0aAMaAVf1S6hdoYQAsDiL6JQZHQ2M
YWr+2KeX4vjnjXKKkKhhiGQ9MeyCNz+X9rnMp63sQvuB+MLa10l0a1SjUvrC
ymEUytKuh05TKJWUWyk5nE9eSyus6jesvHu4F4MxOuUzB6X57sPKWVzBLXiP
xZtGj6q2vafaaT0.BzJfjj.p8ZPukazsQvpfcFfs8mXR3plh8BoBxURIOWyK
rxspZ0YI.eTCEh5Vqp+wGtFXZMKe6CZc3yWZwTdCmYW.BBkdiby8v0r+ST.W
sD9SdUkn8FYspPbqvnBNftZWuYlmlJWo0vuKzeuj2vpJLaWA7Yie7wREui
FpDFDp1KcbAFcP5sJoVxp4NB5Jq40ougIDxJt1wo3GDZHiNocKhiIExx+owv
AdOEAKsDs.RRrOoww1Arc.9RvN2J9tamwjkcqknvAE0l+8WnjHqreNet8whK
z6mukIK4d+Xknv3jstvJs8EirMMhxsZiUsET25jXbX8xczI15xPVxhPcTGFu
xNDu9rXtUCg37g9Q8Yc+EuofIYmg8QdkPCrOnXsaHwYs3rWx9PGs0+pqueG2
uNBqWfhlX7qQG+3.VHcHrf0lNyR2TlqpTM9MDsLKNCQVz6KO.+Sfc5j1Ykj
jzkn2jwNDRP7LVb3d9LtoWBAOnvB92Le6yRmZ4UF7YpQhiFi7A5Ka8zXhKdA
4r9TRGG7V4COiSbAJKdXrWNhhF0hNUh7uBa4Mba0l7JUK+omjDMwkSn95Izr
TOwkdp7W.oPRmNRQsiKeu4j3CkfVgt.NYPEYqMGvvJ48vIlPiyzrIuZskWIS
xGJPcmPiWofLodybH3wjPbMYwlbFIMNHPHFotLBNaLSa9sGk1TxMzCX5Kta6
WIH2ocxSdngM0QPqFRxyPHFsprrhGc9Gy9xoBjz0NWdR2yW9DUa2F85jG2v9
FgTO4Q8qiC7fzzQNpmNpsY3BrYPVJBMJQ1uVmoItRhW9NrVGO3NMNzYZ+zS7
3WTvTOnUydG5kHMKLqAOjTe7fn2bGSxOZDkMrBrGQ9J1gONBEy0k4gVo8qHc
cxmfxVihWz6a3yqY9NazzUYkua9UnynadOtogW.JfsVGRVNEbWF8I+eHtcwJ
+wLXqZeSdWLo+FQF6731Tva0BISKTx.cLwmGJsUTTvkglYsnXmxDge.CDR7x
D6YmX6fMznaF7kdczmJXwm.XSOOrdoHhNA7GMiZYLZZR.+4lconMaJP6JOZ8

```

V1.0

```
ftCs1YWHZI3o.sIXezX5ihMSuXzZtk3ailmXRSczoCS32hAydeyXNEu5SHyS
xqZqbd3ZLderaliPqYxOm++v7SUSz
-----end_max5_patcher-----
*/
```

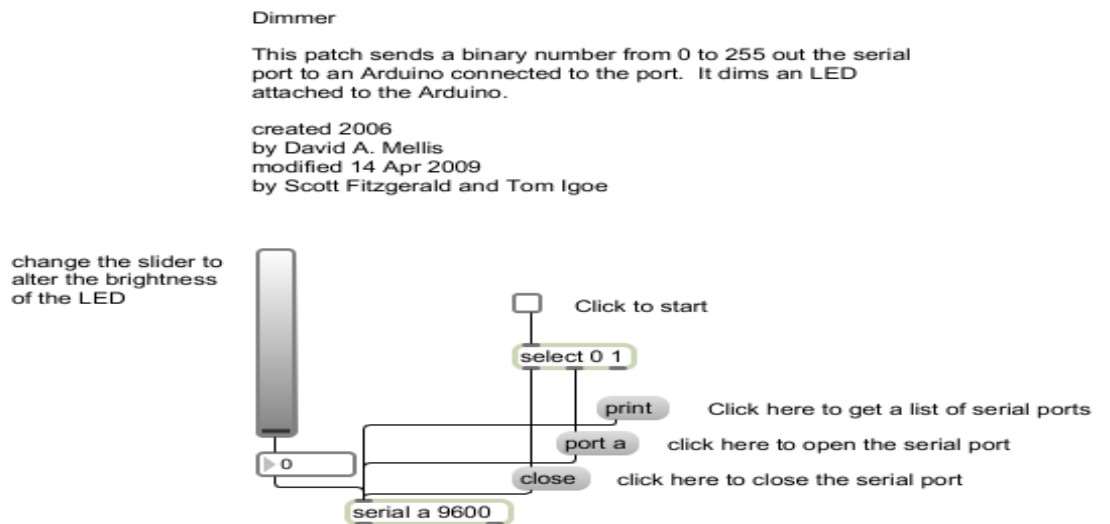
[\[Get Code\]](#)

## Processing Code

The Processing sketch in the code sample above will send bytes out the serial port to the Arduino to dim the LED.

## Max code

The Max/MSP patch in the code sample above looks like the image below. Copy it and paste it into a new patch window.



## Graph

This example shows you how to send a byte of data from the Arduino to a personal computer and graph the result. This is called serial communication because the connection appears to both the Arduino and the computer as a serial port, even though it may actually use a USB cable.

You can use the Arduino serial monitor to view the sent data, or it can be read by Processing (see code below), Flash, PD, Max/MSP, etc.

## Hardware Required

Arduino Board

Analog Sensor (potentiometer, photocell, FSR, etc.)

## Software Required

[Processing](#) or

[Max/MSP version 5](#)

V1.0

## Circuit

Connect a potentiometer or other analog sensor to analog input o.

click the image to enlarge

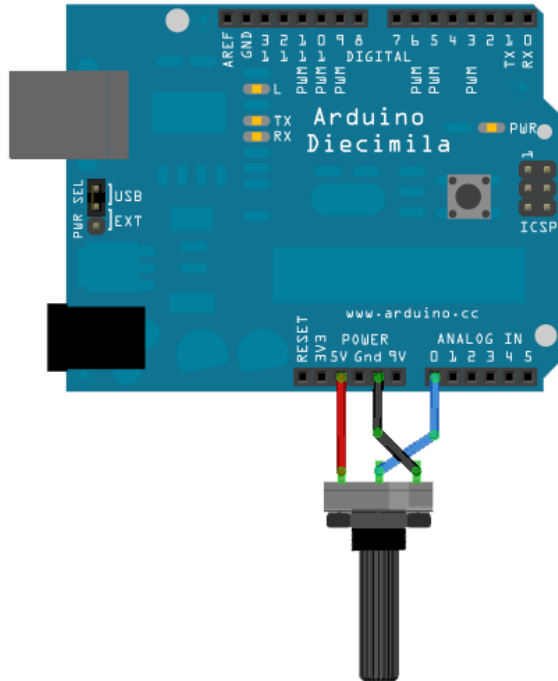
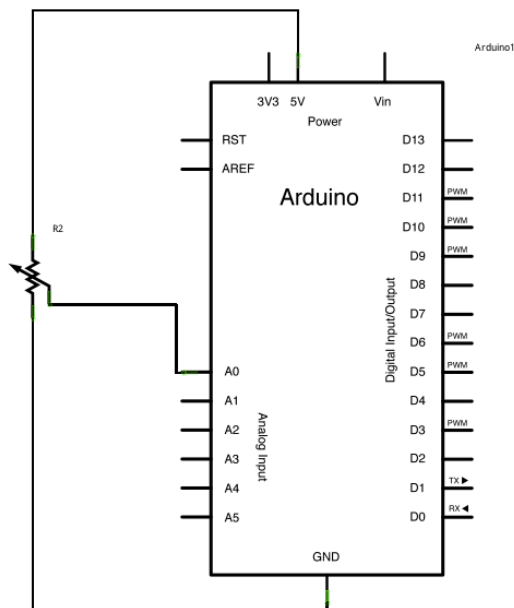


image developed using [Fritzing](#). For more circuit examples, see the [Fritzing project page](#)

## Schematic

click the image to enlarge



## Code

```
/*
```

*Graph*

A simple example of communication from the Arduino board to the computer:

the value of analog input 0 is sent out the serial port. We call this "serial"

communication because the connection appears to both the Arduino and the

computer as a serial port, even though it may actually use a USB cable. Bytes are sent one after another (serially) from the Arduino

to the computer.

You can use the Arduino serial monitor to view the sent data, or it can

be read by Processing, PD, Max/MSP, or any other program capable of reading

data from a serial port. The Processing code below graphs the data received

so you can see the value of the analog input changing over time.

The circuit:

Any analog input sensor is attached to analog in pin 0.

created 2006

by David A. Mellis

modified 9 Apr 2012

by Tom Igoe and Scott Fitzgerald

This example code is in the public domain.

<http://www.arduino.cc/en/Tutorial/Graph>

\*/

```
void setup() {  
  // initialize the serial communication:  
  Serial.begin(9600);  
}  
  
void loop() {  
  // send the value of analog input 0:  
  Serial.println(analogRead(A0));  
  // wait a bit for the analog-to-digital converter  
  // to stabilize after the last reading:  
  delay(2);  
}
```

V1.0

```
}

/* Processing code for this example

// Graphing sketch

// This program takes ASCII-encoded strings
// from the serial port at 9600 baud and graphs them. It expects
values in the
// range 0 to 1023, followed by a newline, or newline and carriage
return

// Created 20 Apr 2005
// Updated 18 Jan 2008
// by Tom Igoe
// This example code is in the public domain.

import processing.serial.*;

Serial myPort; // The serial port
int xPos = 1; // horizontal position of the graph

void setup () {
// set the window size:
size(400, 300);

// List all the available serial ports
println(Serial.list());
// I know that the first port in the serial list on my mac
// is always my Arduino, so I open Serial.list()[0].
// Open whatever port is the one you're using.
myPort = new Serial(this, Serial.list()[0], 9600);
// don't generate a serialEvent() unless you get a newline
character:
myPort.bufferUntil('\n');
// set initial background:
background(0);
}

void draw () {
// everything happens in the serialEvent()
}

void serialEvent (Serial myPort) {
```

V1.0

```

// get the ASCII string:
String inString = myPort.readStringUntil('\n');

if (inString != null) {
  // trim off any whitespace:
  inString = trim(inString);
  // convert to an int and map to the screen height:
  float inByte = float(inString);
  inByte = map(inByte, 0, 1023, 0, height);

  // draw the line:
  stroke(127,34,255);
  line(xPos, height, xPos, height - inByte);

  // at the edge of the screen, go back to the beginning:
  if (xPos >= width) {
    xPos = 0;
    background(0);
  }
  else {
    // increment the horizontal position:
    xPos++;
  }
}

*/

/* Max/MSP v5 patch for this example
-----begin_max5_patcher-----
1591.3oc0YszbaaCD9r7uBL5Ra1QUAO3CvdyS5zVenWZxs5NcfHgJPCIfJIT
RTxj+6AOHkoTDooroUs0AQPR73a+1cwtK3WtZxzEpOwqlB9YveAlL4KWMYh6
Q1GLo99ISKXeJmU451zTUQAWpmNy+NM+SZ2y+sR1102JuU9t0hJvFlNcMPy
dOuBv.U5Rgb0LPpRpYBooM3529latArTUVvzZdFptsXAuDrTU.f.sBffXxL
vGE501IHkUVJXq3fRtdaoDvjYfbgjujaFJSCzq4.tLaN.bi1tJefWpqbo0uz
1IjIABoluxrJ1guxh2JfPO2B5zRNyBCLDFcqbWnvuv9fHCb8bvevyEU2JKT
YhkBSWPAfq2TZ6YhqmuMUo0feUn+rYpY4YtY+cFw3lUJdCMYAapZqzwUHX8S
crjAd+SI0U6UBAwIygy.Q1+HAA1KH6EveWOFQlitUK92ehfal9kFhUxJ3tWc
sgpxadigWExbt1o7Ps5dk3yttivyg20W0VcSmglG90qtx92rAZbH4ez.ruy1
nhmaDPidE07J+5n2sg6E6oKXxUSmc20o6E3SPRDbkXnPGUYE.i5nCNB9TxQ
jG.G0kCTztH88f07Rt0ZMMWUw8VvbKVAaTk6GyoraPdZff7rQTejBN54lgyv
HE0Ft7AvIvvgvIw023jBdUkY0uSvIFSiNcjFhiSsUBwsUCh1AgfNSBAeNDBZ
DIDqY.f8.Yjfv1HAn9XDTxyNFYatVTkKx3kcK9GraZpI5jv7GOx+Z37Xh82
LSKHIDmDXaESoXRngIZQDKVkpXUkMCyXCQhcCK1z.G457gi3TzMz4RFD515F

```

V1.0

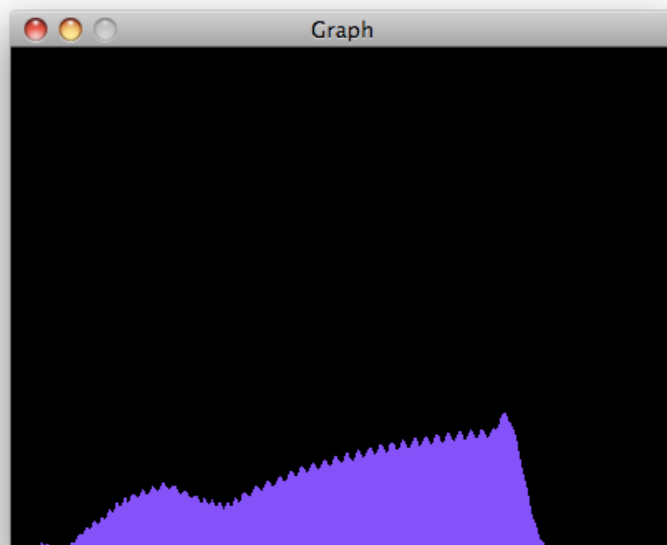
```
G3bIQQwcP3SOF0zlkGhiCBQ1kOHHFFlXaEBQIQnCwv9QF1LxPZ.A4jR5cyQs
vbvHmJsL1l01We+rE2LazX6zYmCraRrsPFwKg1ANBZFY.IAihR8Ox.aH0oAL
hB8nQVw0FSJiZeunOykbT6t3r.NP8.iL+bnwNiXuVMNJH9H9Ycm89CFXPBER
bz422p8.O4dg6kRxdyjDqRwMIHTbT3QFLskxJ8tbmQK4tm0XGeZWf7wKKtYY
aTAF.XPNFaaQBinQMj4QLF0aNHf0JtYuHSxoUZfZY6.UU2ejJTb8lQw8Fo5k
Rv6e2PI+fOM71o2ecY1VgTYdCSxxUqLokuYq9jYJi6lxPgD2NIPePLB0mwbG
YA9RgxdiulK5xiLlSU6JVnx6wzg3sYHwTesB8Z5D7RiGZpXyvDNJY.DQX3.H
hvmcUN4bPlyCkhpTle2P37jtBsKrLWcMScEmltOPv22ZfAQAdKr9HzATQwZ
q18PrUGt6Tst2XMCrUfGuhXs6ccn23YloomMqcTiC5iMGPsHsHRWhWflaenV
XcqwgCQiGGJzptyS2ZMODBz6fGza0bzmXBj7+DA94bvpR01MffAlue07HwcI
pWCwmzJdvi9ILgflLAFmyXB607ML0YbD26lenmcGxjVsZUN+A6pUK7AtTrPg
M+eRYG0qD9j4I7eEbco8Xh6Wco.or9XDC6UCiewbXHkh6xm5LiPEkzpJDRtu
mEB44Fgz4NctJvX.SM1vo2SlTCZGAe7Gzu6ahdRyzFOhYZ+mbVVSyptBw.K1
tboIkatIA7clcTKD1u.honLYV04VklUhsXe0szv9pQCE9Ro3jaVB1o15pz2X
zYoBvO5KXCAe0LCYJybE8ZODf4fV8t9qW0zYxq.YJfTosj1bv0xc.SaC0+AV
9V9L.KKyV3SyTcRtmzi6rO.O16USvts4B5xe9EymDvebK0eMfW6+NIsN1E2m
eqRyJ0utRq13+RjmqYKN1e.4d61jjdsauXe3.2p6jgi9hsNiv97CoyJ01xz1
c3ZhUctSHx3UZgjoEJYqNY+hYs5zZQVFW19L3JDYaTlMLqAA1G2yXlnFg9a
53L1FJVcv.cOX0dh7mCVGCLce7GFcQwDdH5Ta3nyAS0pQbHxegr+tGIZORgM
RnMj5vGl1Fs16drnk7Tf1XOLgvln0d2iEsCxR.eQsNOZ4FGF7whofgfi3kES
1kCeOX5L2rifbdu0A9ae2X.V33B1Z+.Bj1FrP5iFrCYCG5EUWSG.hhunHJd.
HJ5hhnng3h9HPj4lud02.1bxGw.
-----end_max5_patcher-----
```

\*/

[\[Get Code\]](#)

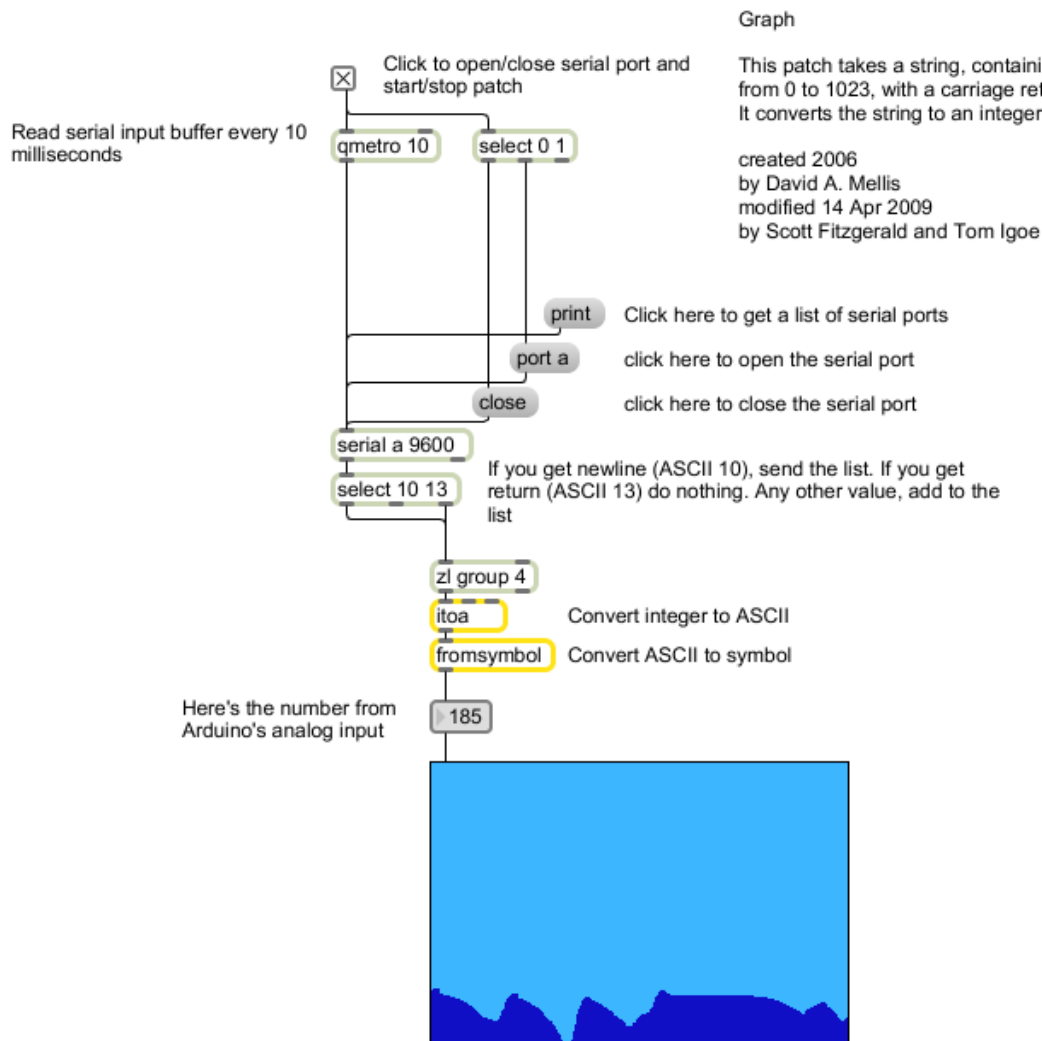
### Processing Sketch

Using the Processing sketch in the code sample above, you'll get a graph of the sensor's value. As you change the value of the analog sensor, you'll get a graph something like this:

**Max Code**

The max patch looks like this. The text of the patch is in the code sample above. Copy the text and paste it into a new Max window to see the sketch.





## Physical Pixel

This example uses the Arduino board to receive data from the computer. The Arduino board turns on an LED when it receives the character 'H', and turns off the LED when it receives the character 'L'.

The data can be sent from the Arduino serial monitor, or another program like Processing (see code below), Flash (via a serial-net proxy), PD, or Max/MSP.

### Hardware Required

Arduino Board

Analog Sensor (potentiometer, photocell, FSR, etc.)

### Software Required

Processing or

Max/MSP version 5

### Circuit

Attach an LED to pin 13. The long leg, or anode, goes to pin 13. The short leg, or cathode, goes to ground. You can also use the built-in LED on most Arduino boards.

[click the image to enlarge](#)

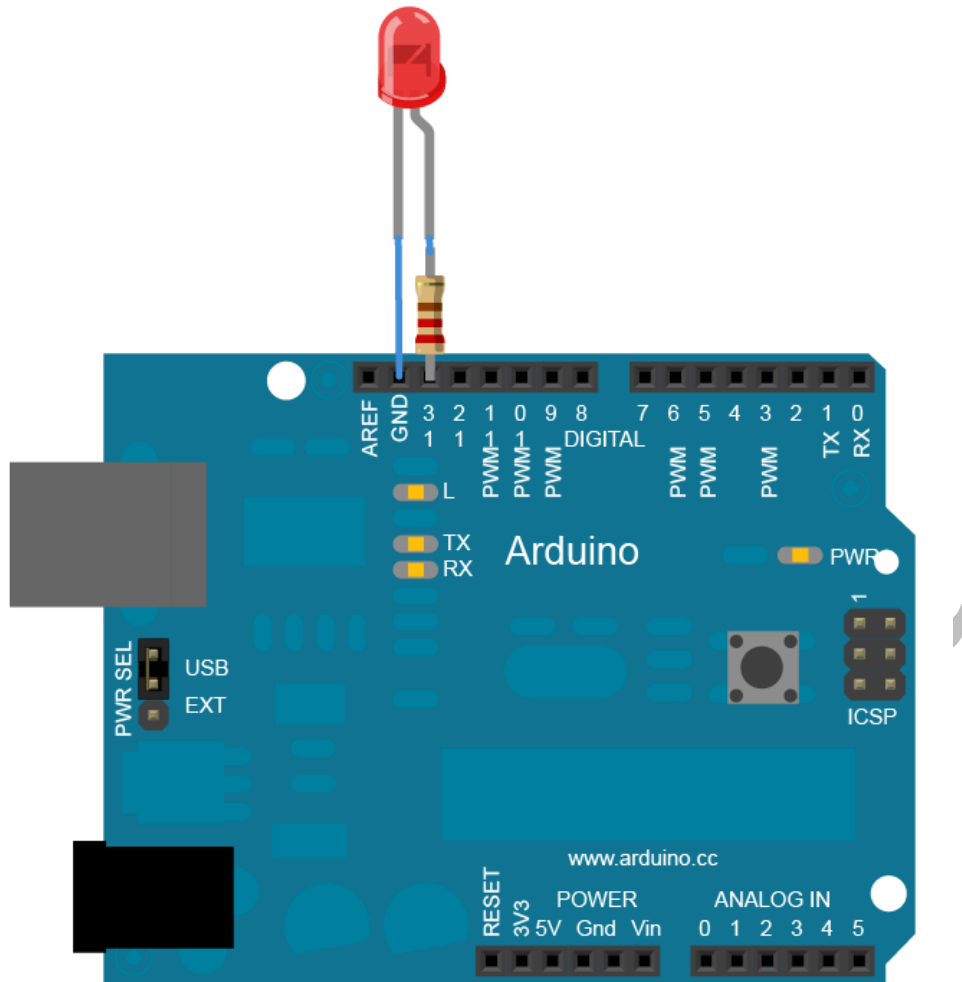


image developed using [Fritzing](#). For more circuit examples, see the [Fritzing project page](#)

## Schematic

click the image to enlarge



## Code

```
/*
```

```
Physical Pixel
```

An example of using the Arduino board to receive data from the computer. In this case, the Arduino boards turns on an LED when it receives the character 'H', and turns off the LED when it receives the character 'L'.

The data can be sent from the Arduino serial monitor, or another program like Processing (see code below), Flash (via a serial-net proxy), PD, or Max/MSP.

The circuit:

\* LED connected from digital pin 13 to ground

*created 2006  
by David A. Mellis  
modified 30 Aug 2011  
by Tom Igoe and Scott Fitzgerald*

*This example code is in the public domain.*

*<http://www.arduino.cc/en/Tutorial/PhysicalPixel>  
\*/*

```
const int ledPin = 13; // the pin that the LED is attached to
int incomingByte;      // a variable to read incoming serial data
into
```

```
void setup() {
  // initialize serial communication:
  Serial.begin(9600);
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // see if there's incoming serial data:
  if (Serial.available() > 0) {
    // read the oldest byte in the serial buffer:
    incomingByte = Serial.read();
    // if it's a capital H (ASCII 72), turn on the LED:
    if (incomingByte == 'H') {
      digitalWrite(ledPin, HIGH);
    }
    // if it's an L (ASCII 76) turn off the LED:
    if (incomingByte == 'L') {
      digitalWrite(ledPin, LOW);
    }
  }
}
```

```
/* Processing code for this example
```

```
  // mouseover serial
```

```
  // Demonstrates how to send data to the Arduino I/O board, in order
  to
```

V1.0

```
// turn ON a light if the mouse is over a square and turn it off  
// if the mouse is not.
```

```
// created 2003-4  
// based on examples by Casey Reas and Hernando Barragan  
// modified 30 Aug 2011  
// by Tom Igoe  
// This example code is in the public domain.
```

```
import processing.serial.*;
```

```
float boxX;  
float boxY;  
int boxSize = 20;  
boolean mouseOverBox = false;
```

```
Serial port;
```

```
void setup() {  
  size(200, 200);  
  boxX = width/2.0;  
  boxY = height/2.0;  
  rectMode(RADIUS);  
}
```

```
// List all the available serial ports in the output pane.  
// You will need to choose the port that the Arduino board is  
// connected to from this list. The first port in the list is  
// port #0 and the third port in the list is port #2.  
println(Serial.list());
```

```
// Open the port that the Arduino board is connected to (in this  
case #0)
```

```
// Make sure to open the port at the same speed Arduino is using  
(9600bps)
```

```
port = new Serial(this, Serial.list()[0], 9600);
```

```
}
```

```
void draw()  
{  
  background(0);  
}
```

V1.0

```

// Test if the cursor is over the box
if (mouseX > boxX-boxSize && mouseX < boxX+boxSize &&
    mouseY > boxY-boxSize && mouseY < boxY+boxSize) {
    mouseOverBox = true;
    // draw a line around the box and change its color:
    stroke(255);
    fill(153);
    // send an 'H' to indicate mouse is over square:
    port.write('H');
}
else {
    // return the box to it's inactive state:
    stroke(153);
    fill(153);
    // send an 'L' to turn the LED off:
    port.write('L');
    mouseOverBox = false;
}

// Draw the box
rect(boxX, boxY, boxSize, boxSize);
}

```

```

*/

/*
Max/MSP version 5 patch to run with this example:

```

```

-----begin_max5_patcher-----
1672.3oc2ZszaaiCD9ryuBBebQVCQRYao8xhf1cQCPVfBzh8RRQ.sDsM2HSZ
Hqmlzh9eu7gjsjsEk7y0oWjiHoHm4aluYHGlueUmtiDuPy5B9Cv8fNc99Uc5
XZR2Pm726zcF4knDRlYXciDylQ4xtWa6SReQZZ+iSeMiEQR.ej8BM4A9C700
kkAlSjQSAYTdbFfvA27o2c6sfO.Doqd6NfXgDHmRUCKkolg4hT06BfbQJGH3
5Qd2e8d.QJIQSow5tzebZ7BFW.FIHow8.2JAQpVIIYByxo9KIMkSjL9D0BRT
sbGHZJikDoZOSMuQT.8YZ5qpgGI3locF4IpQRzq2nDF+odZMIJkrjpEF44M3
A9nWAum7LKFbSOv+PSRXYOvmIhYiYpg.8A2LOUOxPyH+TjPJA+MS9sIzTRRr
QP9rXF31IBZAHpVHkHrfaPRHLuUCzoj9GSoQRqIB52y6Z.tu8o4EX+fddfuj
+MrXiwPL5+9cXwrOVvkbxLpomazHbQO7EyX7DpzXYgkFdF6algCQpkX4XUlo
hA6oa7GWck9w0Gnmy6RXQOoQeCfWwlzsdnHLTq8n9PCHLv7Cxa6PAN3RCKjh
ISRvZ+sS1704Tqt0kocE9R8J+P+RJOZ4ysp6gN0vppBbOTEN8qp0YCq5bq47
PUwfa5e766z7NbGMuncw7VgNRSyQhbnPMGrDsGaFSvKM5NcWoIVdZn44.eOi
9DTRUT.7jdQzSTiF4UzXlc7tLGh4T9pwaFQkGUGIiOOkpBSJUwGsBd40krHQ
9XEvwq2V6eLIhV6GuzP7uzzXBmzsXPSRYwBtVLp7s5lKVv6UN2VW7xRtYDbx

```

V1.0

```

7s7wRgHYDI8YVFatBshkP49R3rYpH3RlUhTQmK5jMadJyF3cYaTNQMGSyHRE
IIUlJaOOukdhoOyhnekEKmZlqU3UkLrk7bpPrpztKBVURluorLddk6xIOqNt
lBORoRrNVFJGLrDxudpET4kzkstNp2lzuUHVmgk5TDZx9GWumnoQTbhXsEtF
tzCcM+z0QKXsngCUtTOEIN0SX2iHTTIIz968.Kf.uhfzUCUuAd3UKd.OKt.N
HTynxTQyjpQD9j1wEXeKQxfHCBahUge6RprSa2V4m3aYOMyaP6gah2Yf1zbD
jVwZVGFZHHxINFxpjr5CiTS9JiZn6e6nTlXQZTAFj6QCpPQwzL0AxVtoi6WE
QXsANkeGWMEEuWnvhmKTnat7A9RqLq6pXuEwY6xM5xRraoTiurj5lJ1vKLzFs
CvM7HI14Mpje6YRxHOSieTsJpvJORjxT1nERK6s7YTN7sr6rylNwf5zMiHI4
meZ4rTYt2PpVettZERbjJ6PjfqN2l0PSrUcush0lCegsGEE5467rnCdqt1ES
QxtCvFq.cvGz+BaAHXKzRSfP+2Jf.KCvj5ZLJRAhwi+SWHvPyN3vXiaPn6JR
3eoA.0TkFhTvpsDMIrL20nAkCI4EoYfSHAuiPBdmJRyd.IynYYjIzMvjOTKf
3DLvnnRLDLpWeEOYXMfAZqfQ0.qsnlUdmA33t8CNJ7MZEb.u7fiZHLyZDkJp
R7CqEVLGN75U+1JXxFUY.xEEBcRCqhOEKz2bENEWnh4pbh0wY25EefbD6EmW
UA6Ip8wFLyuFXx+Wrp8m6iff1B86W7bqJO9+mx8er4E3.abCLrYdA16sBuHx
vKT6BlpIGQIhL55W7oicf3ayv3ixQcm4aQuY1HZUPQWY+cASx2WZ3f1fICuz
vj5R5ZbM1y8gXYN4dIXaYGq4NhQvS5MmcDADy+S.j8CQ78vk7Q7gtPDX3kFh
3NGaAsYBUAO.8N1U4WKycxbQdrWxJdXd10gNIO+hkUMmm.CZwknu7JbNUYUq
0sOsTsI1QudDtjw0t+xZ85wWzd80tMCiiMADNX4UzrcSeK23su87IANqmA7j
tiRzoXi2YRh67ldAk79gPmTe3YKuoY0qdEDV3X8xylCJMTN45JIakB7uY8XW
uVr3PO8wWwEoTW8lSfraX7ZqzZDDXCRqNkztHsGCYpIDDAOqxDPmVUMKcOrp
942acPvx2NPocMC1wQZ8g1Rn3myTykVaEUNLoEeJjVaAevA4EAZnsNgkeyO+
3rEZB7f0DTazDcQTNmdt8aACGi1QOWnMmd+.6YjMHH19OB5gKsMF877x8wsJ
hN97JSnSfLUXGUoj6ujWXd6Pk1SAC+Pkogm.tZ.1lX1qL.pe6PE11DPeMMZ2
.P0K+3peBt3NskC
-----end_max5_patcher-----

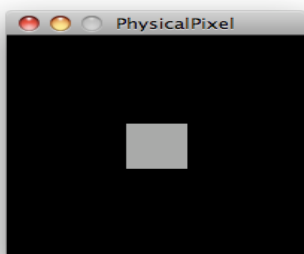
```

\*/

[\[Get Code\]](#)

## Processing Code

Copy the Processing code from the code sample above. As you mouse over the center square, the LED on pin 13 should turn on and off. The Processing applet looks like this:



Mouse over the square to turn the LED on and off.

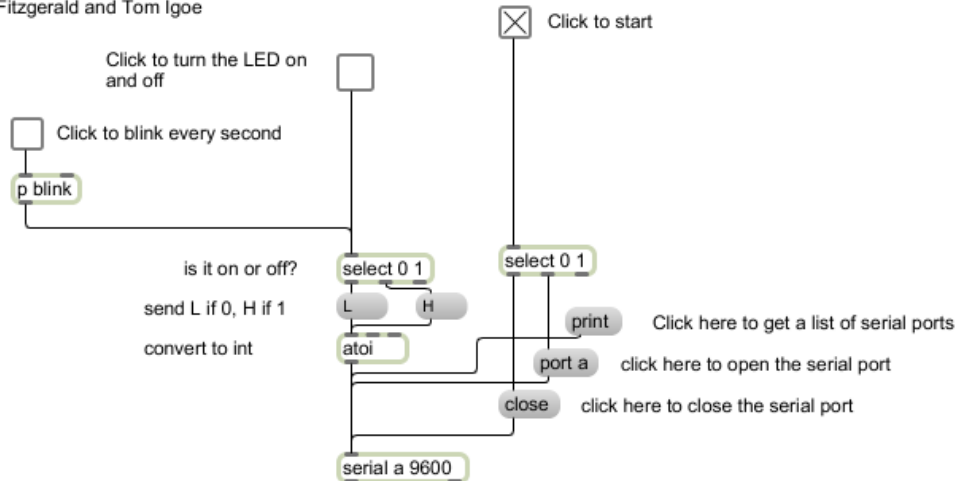
## Max patch

The Max/MSP patch looks like the image below. Copy it from the code sample above and paste it into a new patch window.

## Physical Pixel

This patch sends an ASCII H or an ASCII L out the serial port to turn on an LED attached to an Arduino board. It can also send alternating H and L characters once every second to make the LED blink.

created 2006  
by David A. Mellis  
modified 14 Apr 2009  
by Scott Fitzgerald and Tom Igoe



## Virtual Color Mixer

This example demonstrates how to send multiple values from the Arduino board to the computer. The readings from three potentiometers are used to set the red, green, and blue components of the background color of a Processing sketch or Max/MSP patch.

### Hardware Required

Arduino Board

(3) Analog Sensors (potentiometer, photocell, FSR, etc.)

(3) 10K ohm resistors

breadboard

hook-up wire

### Software Required

Processing or

Max/MSP version 5

### Circuit

Connect analog sensors to analog input pins 0, 1, and 2.

This circuit uses three voltage divider sub-circuits to generate analog voltages from the force-sensing resistors. a voltage divider has two resistors in series, dividing the voltage proportionally to their values.

Click on the image to enlarge

V1.0

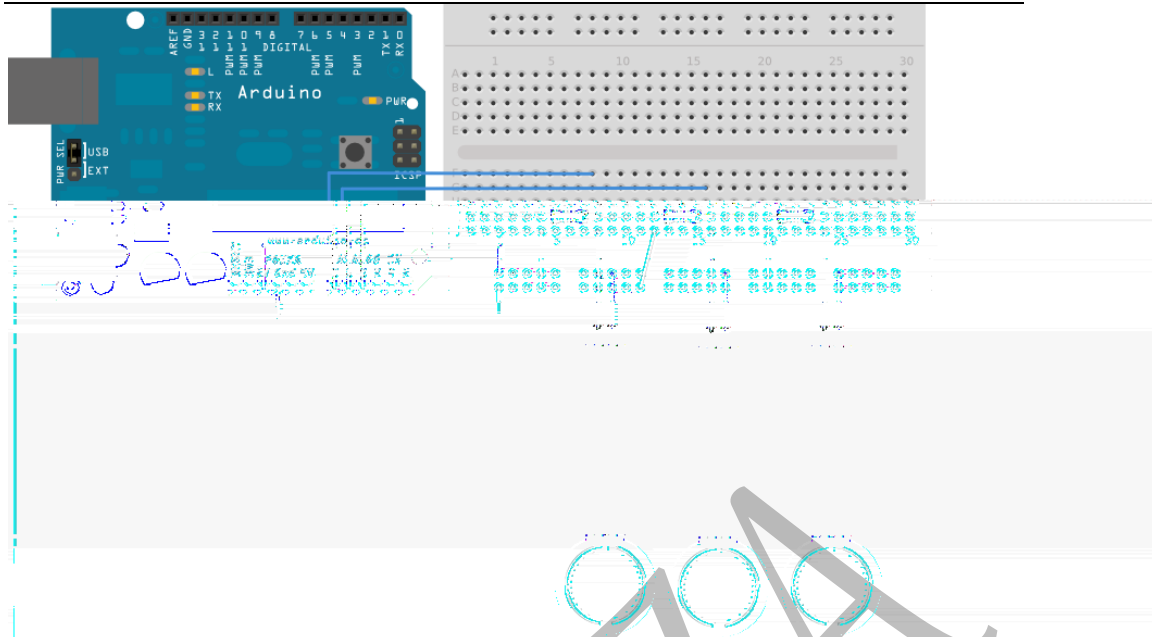
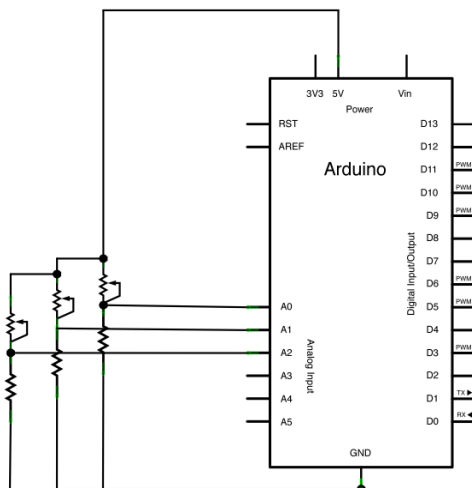


image developed using [Fritzing](#). For more circuit examples, see the [Fritzing project page](#)

## Schematic

Click on the image to enlarge



## Code

The sensor values are sent from the Arduino to the computer as ASCII-encoded decimal numbers. This means that each number is sent using the ASCII characters "0" through "9". For the value "234" for example, three bytes are sent: ASCII "2" (binary value 50), ASCII "3" (binary value 51), and ASCII "4" (binary value 52).

```
/*
```

```
  This example reads three analog sensors (potentiometers are
  easiest)
```



V1.0

*and sends their values serially. The Processing and Max/MSP programs at the bottom*

*take those three values and use them to change the background color of the screen.*

*The circuit:*

*\* potentiometers attached to analog inputs 0, 1, and 2*

*<http://www.arduino.cc/en/Tutorial/VirtualColorMixer>*

*created 2 Dec 2006*

*by David A. Mellis*

*modified 30 Aug 2011*

*by Tom Igoe and Scott Fitzgerald*

*This example code is in the public domain.*

*\*/*

```
const int redPin = A0;      // sensor to control red color
const int greenPin = A1;    // sensor to control green color
const int bluePin = A2;     // sensor to control blue color
```

```
void setup()
```

```
{
```

```
  Serial.begin(9600);
```

```
}
```

```
void loop()
```

```
{
```

```
  Serial.print(analogRead(redPin));
```

```
  Serial.print(",");
```

```
  Serial.print(analogRead(greenPin));
```

```
  Serial.print(",");
```

```
  Serial.println(analogRead(bluePin));
```

```
}
```

```
/* Processing code for this example
```

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

```
import processing.serial.*;
```

```
float redValue = 0;      // red value
```

```
float greenValue = 0;    // green value
```

V1.0

```
float blueValue = 0;          // blue value

Serial myPort;

void setup() {
  size(200, 200);

  // List all the available serial ports
  println(Serial.list());
  // I know that the first port in the serial list on my mac
  // is always my Arduino, so I open Serial.list()[0].
  // Open whatever port is the one you're using.
  myPort = new Serial(this, Serial.list()[0], 9600);
  // don't generate a serialEvent() unless you get a newline
  character:
  myPort.bufferUntil('\n');
}

void draw() {
  // set the background color with the color values:
  background(redValue, greenValue, blueValue);
}

void serialEvent(Serial myPort) {
  // get the ASCII string:
  String inString = myPort.readStringUntil('\n');

  if (inString != null) {
    // trim off any whitespace:
    inString = trim(inString);
    // split the string on the commas and convert the
    // resulting substrings into an integer array:
    float[] colors = float(split(inString, ","));
    // if the array has at least three elements, you know
    // you got the whole thing. Put the numbers in the
    // color variables:
    if (colors.length >= 3) {
      // map them to the range 0-255:
      redValue = map(colors[0], 0, 1023, 0, 255);
      greenValue = map(colors[1], 0, 1023, 0, 255);
      blueValue = map(colors[2], 0, 1023, 0, 255);
    }
  }
}
```

V1.0

\*/

/\* Max/MSP patch for this example

```

-----begin_max5_patcher-----
1512.3oc4Z00aaaCE8YmeED9ktB35xOjrj1aAsXX4g8xZQeYoXfVh1gqRjdT
TsIsn+2K+PJUovVVJ1VMdCAvxThV7b07b48dIyWtXxzksaYkSA+J3u.Sl7kK
1LwcK6MlT2dxzB5so4zRW2lJXeRt7elNy+HM6Vs61uDDzbOYkNmo02sg4euS
4BSede8S2P0o2vEq+aEKU66PPP7b3LPHDauPvyCmAvv4v6+M7L2XXF2WfCaF
1URgVPKbCxzKUbZdySDUEbgABN.ia08R9mccGYGn66qGutNir27qWbg8iY+7
HDRx.Hjf+OPHCQgPdpQHoxhBlwB+QF4cbkthlCRk4REnfeKScs3ZwaugWBbj
.PS+.qDPAKZkgPlY5oPS4By2A5aTLFv9pounjsqpnZVF3x27pqtBrRpJnZaa
C3WxTkfUJYA.BzR.BhIy.ehquw7dSoJCsrlATLckR.nhLPNWvVwL+Vp1LHL.
SjMG.tRaG7OxT5R2c8Hx9B8.wLCxVaGI6qnpj45Ug84kL+6YIM8CqUxJyyCF
7bqsBRULGvwfWyRMyovElat7NvqoejaLm4f+fkmyKuVTHy3q3ldhB.WtQY6Z
x0BSOeSpTqA+FW+Yy3SyybH3sFy8p0RVCmaMpTyX6HdDZ2JsPbfSogbBMueH
JLd6RMBdfRMzPjZvimuWIK2XgFA.ZmtfKohQSm88qc6OF4bDQ3P6kEtF6xej
.OkjD4H5O1lyS+.3FlhY0so4xRlWqyrXERQpt+2rsnXgQNZHZgmMVzEofW7T
S4zORQtgIdDbRHR0bRzSMNoFUVZVcbKbhQZrSOo934TqRHIN2ncr7BF8TKR1
tHDqL.PejLRRPKMR.pKFAkbtDa+UOvsYsIFH0DYsTCjqZ66T1CmGeDILLpSm
myk0SdKOKh5LUR4GbWwRYdW7fm.BvDmzHnSdH3biGpSbxxDNJoGDADlChH7L
I0DalOoTBLvk07zPs5HJnKNogAXbol5eytUhfyiSfnjEluAq+Fp0a+wygGwR
q3ZI8.psJpkpJnyPzwmXBj7Sh.+bNvVZxlcKAm00YHIxcIjzEKdRChgO5UMf
LkMPNNOMfiS7Ev6TYQct.F5IWCCZ4504rGsiVswGWWSYyma01QcZgmL+f+sf
oU18Hn6o6dXkMkFF14TL9rIAWE+6wvGV.p.TPqz3HK5L+VxYx14UmBKEjr.B
6zinuKI3C+D2Y7azIM6N7QL6t+jQyZxymK1ToAKqVsxjlGyjj2c1kTK3180h
kJEYkacWpv61yp2VJTjWK47wHA6fyBOWxH9pUf6jUtZkLpNKW.9EeUBH3ymY
XSQlaqGrkQMGzp20adYSmIOGjIABolxZyAWJtCX9tg6+HMuhMCPyx76ao+Us
UxmzUE79H8d2ZB1m1ztbnOalmGeAq0awyK8a9UqBUc6pZolpzurTK232e5gp
aInVw8QIIcpaiNSJfy4Z+92Cs+Mc+mvg2cEsvGLLY6V+1kMuioxnB5VM+fsY
9vSu4WI1PMBGXye6KXvNuzmZTh7U9h5j6vvASdngPdgoFxyCNL6ialaxUMmT
JIzebXcQCn3SKMf+4QCMmOZung+6xBCPLfw08ngcEI52YJ1y7mx3CN9xKUYU
bg7Y1yXj1KW6SrZnguQdsSfOSSDItqv2jwJFjavclv07OigyBr2+gDYorRk1
HXZpVFfu2FxXkZtfp4RQqNkX5y2sya3YYL2iavWAOaizH+pw.Ibg8f1I9h3Z
2B79sNeOHvBOTfEalWsvyu0KMf015.AaROvZ7vv5AhnndfHLbTgjcCK1K1Hv
gOk5B26OqrXjcJ005.QqCHn8fVTxnxfj93SfQiJlv8YV0VT9fVUwOOhSV3uD
eeqCUClbBPa.j3vWDoMZssNTzRNEE6gYPXazZaMF921syaLWyAeBXvCESA8
ASi6Zyw8.RQi65J8ZsNx3ho930hGWENTWpowepae4YhCFELerOLEntXJrOSc
iadi39rf4hwc8xdhHz3gn3dBI7iDRlFe8huAfIZhq
-----end_max5_patcher-----

```

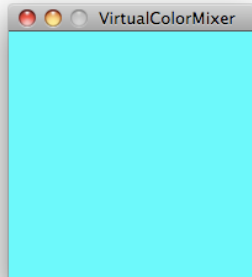
\*/

[\[Get Code\]](#)

V1.0

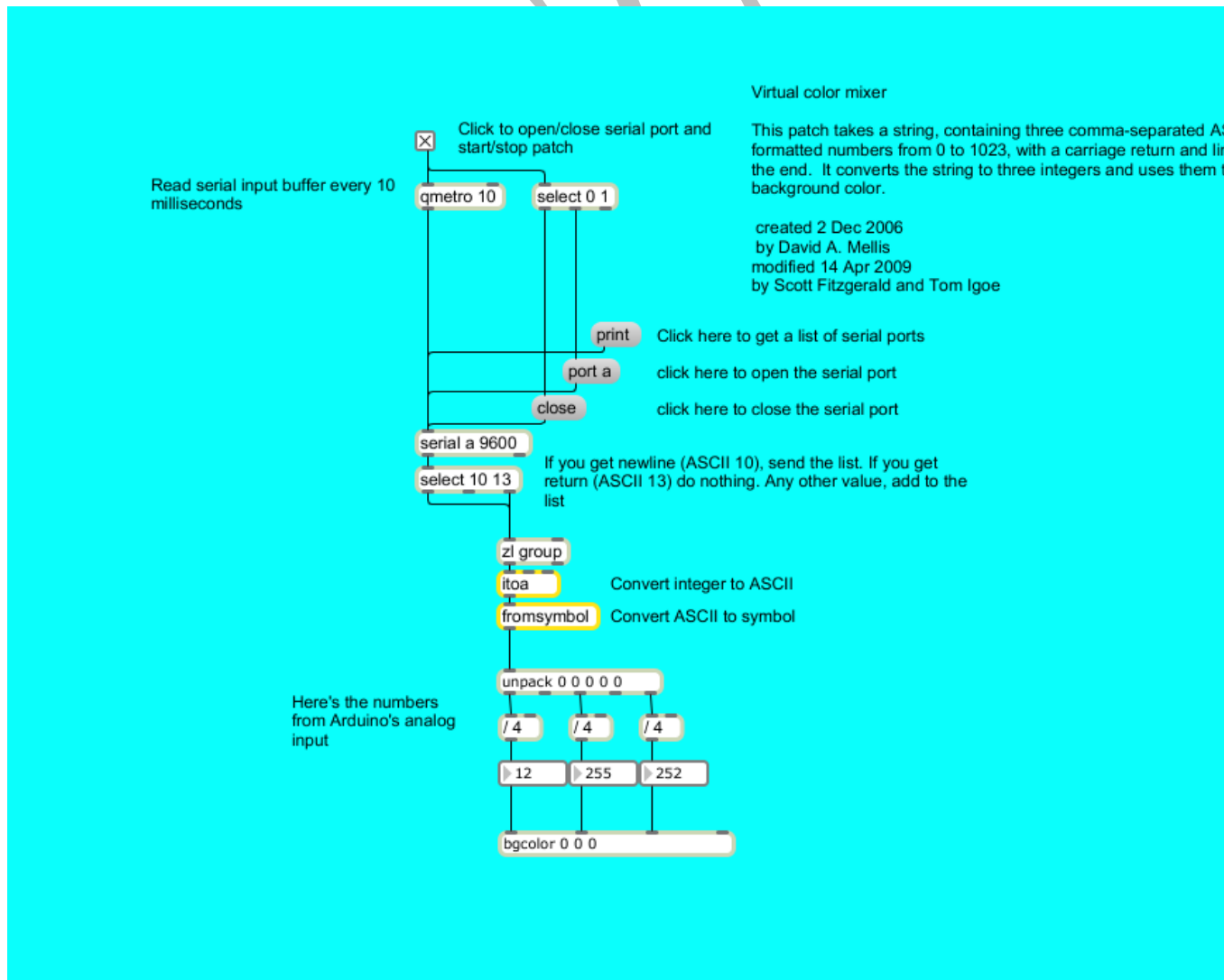
## Processing Code

Copy the Processing sketch from the code sample above. As you change the value of the analog sensors, the background color will change:



## Max Code

The max patch looks like this. Copy the text of it from the code sample above and paste into a new Max window.



## Serial Call and Response (handshaking)

This example demonstrates multi-byte communication from the Arduino board to the computer using a call-and-response (handshaking) method.

This sketch sends an ASCII A (byte of value 65) on startup and repeats that until it gets a serial response from the computer. Then it sends three sensor values as single bytes, and waits for another response from the computer.

You can use the Arduino serial monitor to view the sent data, or it can be read by Processing (see code below), Flash, PD, Max/MSP (see example below), etc.

### Hardware Required

Arduino Board

(2) analog sensors (potentiometer, photocell, FSR, etc.)

(1) momentary switch/button

(3) 10K ohm resistors

breadboard

hook-up wire

### Software Required

Processing or

Max/MSP version 5

### Circuit

Connect analog sensors to analog input pin 0 and 1 with 10K ohm resistors used as voltage dividers. Connect a pushbutton or switch to digital I/O pin 2 with a 10Kohm resistor as a reference to ground.

click on the image to enlarge

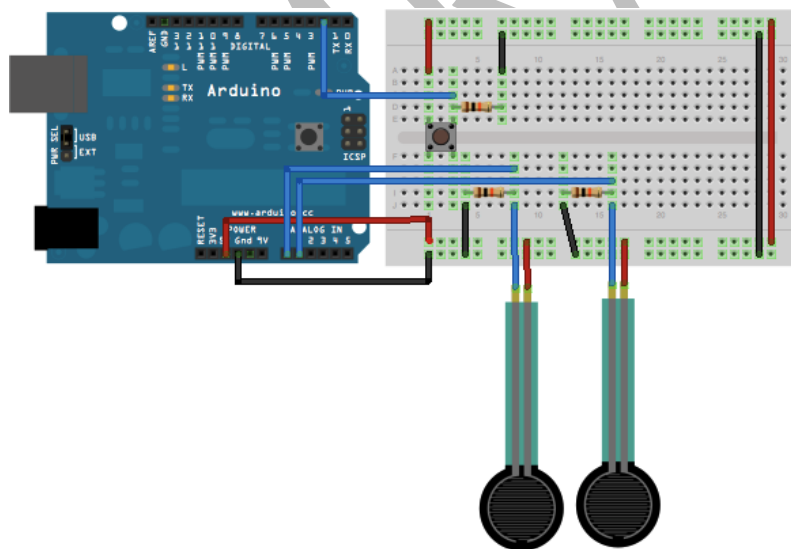


image developed using [Fritzing](#). For more circuit examples, see the [Fritzing project page](#)

### Schematic

click the image to enlarge

### Code

```
/*  
  
Serial Call and Response
```

V1.0

*Language: Wiring/Arduino*

*This program sends an ASCII A (byte of value 65) on startup and repeats that until it gets some data in.*

*Then it waits for a byte in the serial port, and sends three sensor values whenever it gets a byte in.*

*Thanks to Greg Shakar and Scott Fitzgerald for the improvements*

*The circuit:*

- \* potentiometers attached to analog inputs 0 and 1*
- \* pushbutton attached to digital I/O 2*

*Created 26 Sept. 2005*

*by Tom Igoe*

*modified 24 April 2012*

*by Tom Igoe and Scott Fitzgerald*

*This example code is in the public domain.*

*<http://www.arduino.cc/en/Tutorial/SerialCallResponse>*

```

*/

int firstSensor = 0;    // first analog sensor
int secondSensor = 0;  // second analog sensor
int thirdSensor = 0;   // digital sensor
int inByte = 0;        // incoming serial byte

void setup()
{
    // start serial port at 9600 bps:
    Serial.begin(9600);
    while (!Serial) {
        ; // wait for serial port to connect. Needed for Leonardo only
    }

    pinMode(2, INPUT); // digital sensor is on digital pin 2
    establishContact(); // send a byte to establish contact until
    receiver responds
}

void loop()
{

```

V1.0

```

// if we get a valid byte, read analog ins:
if (Serial.available() > 0) {
  // get incoming byte:
  inByte = Serial.read();
  // read first analog input, divide by 4 to make the range 0-
255:
  firstSensor = analogRead(A0)/4;
  // delay 10ms to let the ADC recover:
  delay(10);
  // read second analog input, divide by 4 to make the range 0-
255:
  secondSensor = analogRead(1)/4;
  // read switch, map it to 0 or 255L
  thirdSensor = map(digitalRead(2), 0, 1, 0, 255);
  // send sensor values:
  Serial.write(firstSensor);
  Serial.write(secondSensor);
  Serial.write(thirdSensor);
}
}

void establishContact() {
  while (Serial.available() <= 0) {
    Serial.print('A'); // send a capital A
    delay(300);
  }
}

/*
Processing sketch to run with this example:

// This example code is in the public domain.

import processing.serial.*;

int bgcolor;           // Background color
int fgcolor;           // Fill color
Serial myPort;         // The serial port
int[] serialInArray = new int[3]; // Where we'll put what we
receive
int serialCount = 0;    // A count of how many bytes
we receive
int xpos, ypos;        // Starting position of the ball
boolean firstContact = false; // Whether we've heard from

```

V1.0

*the microcontroller*

```

void setup() {
    size(256, 256); // Stage size
    noStroke();      // No border on the next thing drawn

    // Set the starting position of the ball (middle of the stage)
    xpos = width/2;
    ypos = height/2;

    // Print a list of the serial ports, for debugging purposes:
    println(Serial.list());

    // I know that the first port in the serial list on my mac
    // is always my FTDI adaptor, so I open Serial.list()[0].
    // On Windows machines, this generally opens COM1.
    // Open whatever port is the one you're using.
    String portName = Serial.list()[0];
    myPort = new Serial(this, portName, 9600);
}

void draw() {
    background(bgcolor);
    fill(fgcolor);
    // Draw the shape
    ellipse(xpos, ypos, 20, 20);
}

void serialEvent(Serial myPort) {
    // read a byte from the serial port:
    int inByte = myPort.read();
    // if this is the first byte received, and it's an A,
    // clear the serial buffer and note that you've
    // had first contact from the microcontroller.
    // Otherwise, add the incoming byte to the array:
    if (firstContact == false) {
        if (inByte == 'A') {
            myPort.clear();          // clear the serial port buffer
            firstContact = true;      // you've had first contact from the
microcontroller
            myPort.write('A');        // ask for more
        }
    }
    else {

```



V1.0

```

// Add the latest byte from the serial port to array:
serialInArray[serialCount] = inByte;
serialCount++;

// If we have 3 bytes:
if (serialCount > 2 ) {
    xpos = serialInArray[0];
    ypos = serialInArray[1];
    fgcolor = serialInArray[2];

    // print the values (for debugging purposes only):
    println(xpos + "\t" + ypos + "\t" + fgcolor);

    // Send a capital A to request new sensor readings:
    myPort.write('A');
    // Reset serialCount:
    serialCount = 0;
}
}
}
*/

/*
Max/MSP version 5 patch to run with this example:

-----begin_max5_patcher-----
3908.3oc6ckziiaIE9b0+J3XjCIXpp.WzZNMURv.jCInQ5fYNjNngrDssRKK
4nkp6JA4+973hrkrsjncKu0SRiXasQ83G+dKj7QV+4qtaxzrOxKlf9Zzufft6
t+7U2cm7ThSbm936lrL3igIAExaaRJ+CYS+sI2qtTI+ikxSuBMKNojm+N3D4
Aua5KkPwpuoUAKgKhSm+tbDxo5cQXVOhuGwrohuHD4WT7iXzupen3HY4BuqG
rH0kzrrzxzfbk4kdJONHo9JoUKiSS3kRgjt4jYUk0mkznPJh+CYgHewpSqty
xWVwUh3jIqkEYEFmqQEMr.ETbB+YddQbVZix+tIAqV03z203QDX4ukIKHm6W
ep3T0ovqOUN+435m2Rcx+5U0E+FTzVBh9xOsHXIh5YuADg1x4IYgumG0r3mj
shmFmtJmWvSKCJ0um0WNhOKnJo7c6GmZe8YAg7Ne381Rc2j44wQYoBgn0SJN
c8qCHH1RhQqJi7NRCVsmGt.pGUESCxE31zDdCV.PRyXRZeo0MU.WOHMdYPIu
LVIrT75BMd4p73zxVuHdZ.TFKJByyRRZUTpq77dtRDzZFx+PbT4BYy0DJgaO
dUcSvj0XTT7bdQY6yUFLun8YZo71j10TIt042RYNLa4RfCTWfsznKWDWfJpl
tJHrbgV6t.AZInfzWP.4INpJHA8za91u+6QN1nk7hh.PpQwonxEbTAWzpilV
MimilkmsDtPbo3TPiUdY0pGa9ZShS4gYUJz1pwEliwCpxbAgJI9DGGwWNzFT
ksLf3z7M0MybG6Hj1WngsD7VEXS8j5q7Wu5U0+39ir8QJJS5GMHdtRimL4m1
0e1EVX0YsE2YssINriYRoFRyWVMoRRUGQvnkmms3pnXDYHbBKMPpIOL5ils8
3rMPwFcRCsGRyPH780.8HBnpWz.v1EQBWJ+0CSunehJSmJxiIZRtNGhhDYrU
jt3ZQyA2fHJhZDifXIQHUHH8oGYgOREI5nqHIzhFWUndPyBdB3VzHJGwUhkV
rgvRl2UCVNMHcd234lf1DN16HFEIdHt99A5hrp7v5WWMSBQZgMP.Tkwoqig8

```

V1.0

W1.Sn1f3h3nn1wLpBypPDz1J7XinEGkLiMPloWOhrgR7dpZWJQV1faDy35Qj  
MThMFkWFGsJChQPqrQp8iorV6Q28HBVF4nMVDJj7f1xyYACFScisg.ruLHOW  
uMUS4Am4pI4PTnHi.6bi02HNzSYnDBe4cgAgKzRk1jc8PJLoH3Ydz6.Q.7K8  
tfxx73oUkJq1MGuCy5TpAi.POWZ3AenidLOOIaZPhdjZVW3sdk6LXEGzHb7p  
Mfr7SEy3SXHyBSxJ3J2ncNNYVJsXG6Me10nj4cfCRFdTFjLo7q3SiCpjJEDM  
.nvra.GN39.E2CDTHWXPo8.xzfqrHCHKnf5QUYUVdoZPUjCSC7LU8.XtTUX1  
X8vr51GjwFGLC2AlMdLkU4RiaRrnmJuiudnDk0ZW+9p6TuKBe433JUCzp6fU  
iOF0Suk2UQYUPNTEkiZubvKa1tsmgL5SCTXGHnnG0CceLpkpR9Rs28IUESW1  
EwWNKfHlg.zj6Ee7S+nE8A+m9F7Cu40u9gMm+aRp3kYYkKd3GDOz5y+c7b96  
K9gfvuIK68uNO6g2vUUL80WxihCVFD9v1B30e2SOrmXUb527RZ3nZNrljGrR  
70vslJ9suWuZ3zaHvDg3RIJLgGj2Gfn6TcGcstEfvth.hpFLlnBndjOLGQAI  
z98BXC6yQxghmOn6gZqj0ShPOXhynLojzCESt+XwE8TxrCvrdXo16rqnlGvb  
HaFmbh29QD+K0DyNdjDwvzQL.NXpoMvoOBxkger0HwMRQbpbCh91fjjG9Idw  
prTH9SzaSea5a.GQEPnnh43WNefMlsOgx18n.vgUNO.tKl7tDyI3iHzaFJHZ  
VVNedVEbGgYIY42i93prB0i7B7KT1LnnCiyAinPbnsPV7OG.tYKfBsrJOkg  
UG5aq26iJw6GyJ4eM5mEgEKaNQPMEBUp.t8.krp1OVT1ZdJAW27bjvGK7p2p  
HQPgLOSJDYv4E9gQBYBjMUselRxDy+4WplIzm9JQAWOEmfb.E364B43CAwp5  
uRRDEv8hWXprjADMUOYpOg9.bVQpEfHkgGCnAnk.rghBJCdTVICA3sDvAhE5  
oU4hf67ea5zWPuILqrD8uiK+i477fjHIt9y.V88yy3uMsZUj7wnxGKNAdPx5  
fAZMErDZOcJU4M01WFQokix.pKa+JE1WacmnKFeYd7b.0PeIzB8Kk+5WIZpB  
Ejt34KJeHgOch4HK8Y3QiAkAfs8TRhhOkG7AAGQf0qxyfmQxa+PLb8Ex.2PS  
4Bd05GB9Hvg+cfJCMofAlMu9Qz+UPCjckqVJlEmyA8Bf.rc6.3hAEuG8TdTU  
bZljQ0nr1ayIqmTwQYfyRGafZhur5vfuyMSqYNWmtAPwWHalDSuUgT0Bosh.  
JpAR89Y6Ez5QEfPTQO4J0DHLInI1iz8BZV2JfV3Bd36qsQwAVVXbr1BGXp6s  
Sd5sSDruo74wofx.HxUgxQwTnMLqTXvRmiGh2PUZr5pBynKChj16feNUjSRn  
hEUfRPT1GfG9Ik4TQBM.hEZZ.bc38HjAMKGzDRijEm1ifx1dbgzQyKh6FZc3  
wOckRJH+KU0daWs6wzltWx1puXxlWW6NZWY2JiTbzzILRIANku02NourySM  
VI1VJTvQZff32AJr+dS9e34QAoA6EGX1GFH9yk7yyQAlVd3SR94g+TxOulsU  
Flgd6ICI96LzazyPulcgqsZ8r74SgF.65+efbMf4pGHT7lgHh30Sha3N5Ia.  
oqjMf7nsuMwycf7iYDybiAAVr3eC.oTMjPzEr8GDRc9bFRGHYXDrzg.Tlx+q  
NW8TY1IkzCfZ2IftkQstbB08HUEzoDS+oFyI.cWIhWBaDiUo7qIrDO7f.L6n  
AXqCmyNT9act.z+Iv.GR0uES0ZXfjdz.IczAxQOUR+zvRsUTigRxmyPYeNlj  
yXv8Peef2ZFzuLzWPPeAE8ELzWXYlhe8WzAcUg+b1UkIoCLzIH60zwASGXau  
a1Dq2nUY.sox4vng+m0nACePngC91EMLZMBPodOxf+yx5d4uMCTHm3kJvIIG  
jclMedEQldkjpoBkQyY1Hk.hmSY95Iwos8NDb9VSlIWOIntqgxryUjL6bCJ  
y111i5tWWxrQ7YmqGYlc6shKliY2dr0wtNjYxgHyzaq0OznY235awCr8zSz6  
EGd1QNUKf.74dADTBbTbeotjpW95Io1Y0WpKYONY8M83Rx2MChx3fL+iG5Mm  
tXpdmvXj8uTvaAL1WjbbarQD4Z6kXBpnm6a69oKV2PY9WY174IbC3CaRQ9iK  
Q4sYGQpWdtZ5wFrc7n569.M83OOR5ydsB1ZcAWCxdbKuavz9LILxFeD.WWO.W  
Nq+Zu4Es+AP6s5p9jDWH8ET+c85+XbW0.N1nDCTD7U4DGc6ohnU019fS7kQ0  
o431uuOGjv5agHp0DT.CysOfgLR3xX1XTUKm16RivRsn3z006c13YScAvtrb  
hwekGB7BZuqESUZBJWmCvK7t9HF8Ts6cUAPoFWso3aP8ApWyJ3wqOPo2pJDC  
BQ0NI0Pj8QCQ2r1L5vKaU51DRYX7yRur1UYYZmJQ9iDHwN9dndB5n5ejflmm  
UsBwLHnDkKXRuAkb3NeuzqRstiQGP.fcQFdHNzaE.8u58Nz9svFE9SGIE1X

V1.0

```

kv9Iwfl1BdNWjA7xcThsWCS847loyFD8pZq2E2F04lYULzBTDYhrFSDDJdjo
fisN2NUN26e4xRu51zD5ZseJ4HC63WyIX6jRqsp0jangBnK.Qlo58PCpWewt
ahzqK7fbKsdX6R64aao8LmWhBPh9jKVAPMzb5a2cV6opdWHneMmqMEmAGsPh
ieigIjV+4gF1GgbMNXg+NH44YaRYyd..S1ThHzKhFwwGRaWVITqyj9FvPqMT
d0pDuSgDrOGF.Uogf.juCFi9WAUkYR+rFPanDcPG8SbrtjyG03ZQ8m3AqC5H
NcUUoXSwVrqXKVCZu.5ZnkwIfIVdXVZTwAuTTUiYuxwjZDK6ZgnRtYV8tJmP
hEcuXgz2Goxyaiw35UkaWbpqtfzD02oUkkYqi.YQbZqIIWrIljFolsdmMKFR
wCJ2+DTn.9Qlk0ld+d9Qy9IJdpLfy05Ik2b8GsG9h8rdm1ZFxlFrmmlA2snw
qI9Mcdi2nr6q3Gc87nLawurbwldda+tmYgJ9HaQmlkGwy6davisMgrkM65oz
eulFYCzG46am8tSDK144xV4cEvVMTRXq9CIX8+ALNWb6sttKNkiZetnbz+lx
cQnblNds2C0tvLNe14hwQtXybxhqc17qHfamUcZZ3NYSWqjJuiDoizZ+ud2j
naRK4k3346IIVdR1kKiQjM39adMamvc6n+Xp36Yf3SIGH3uKbquqs1JksTII
kuJ7RrZSFb2Cn9j5a6DT8cMo0iczU+lsYaU8YNVh5k5uzJLU26ZcfuJE6XLY
0mcRp9NTCp+L+Ap+in7Xf3b9jFQBLtIY06PbrGhcrU6N00Qlaf9N0+QP09nS
P6qsI7aYNLSNOHpsAxis0ggnZLjYqyyFkdSqinVsPaqSDZaYBZ6c93uLCjGm
iCroJVLzU45iNE.PIUfs3TWb.0FejHp9uANr0GcJPTroFDNOHpkIweLnI1QT
dHl3P7LhOF3Ahd9rnlwAMy5JSdNezGlsIsW9mW44r26js+alhxlkdhN0YE
YqiH5MTeWo6D4Qm.ieLS7OynmuVGSbmbFUlnWWhiQlhOeN+Yl35bq.tGo9JR
cj8AVqdz7nSgVB9zNj.FTOU68o5d9KO5TUOGxVMw+jTO8T6wqD0hEiHsOJO5
TTOMoS.zlqN0SpZjz6GcH05ylVM0jwuidlkmAif374ih5M5QPfCCR8Hqifff
otN8pt3hUcaWu8nosBhwmD0Epw5KmoF.poxy4YHbnjqfPJqcM3Y2vun7nS.i
f3eETiqcRX2LR.4QmhZrkoCSGwzZrQKhrVR8caari+55d2caPqmq5n.ywe8Q
WrZL9fpwVXeaogMBY6y1SMdjk+gbavbN7fYvVtt1C2XwHJSzpk+tidU025H
UB9onw9mlFQ10fhpZBaDatcMTTEGcJp wzqg92qqiVtM6Cu0IRQ0ndEdfCAqV
10qYAUmPrctbx04XCuPMalasYzKDks1D52ZCne6Mednz9qW8+.vfqkDA
-----end_max5_patcher-----

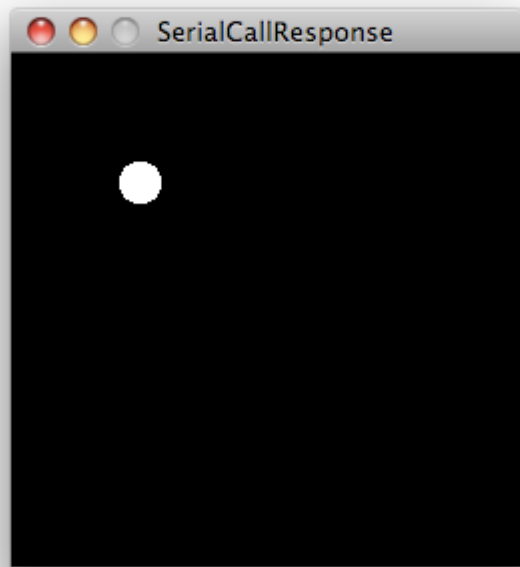
```

\*/

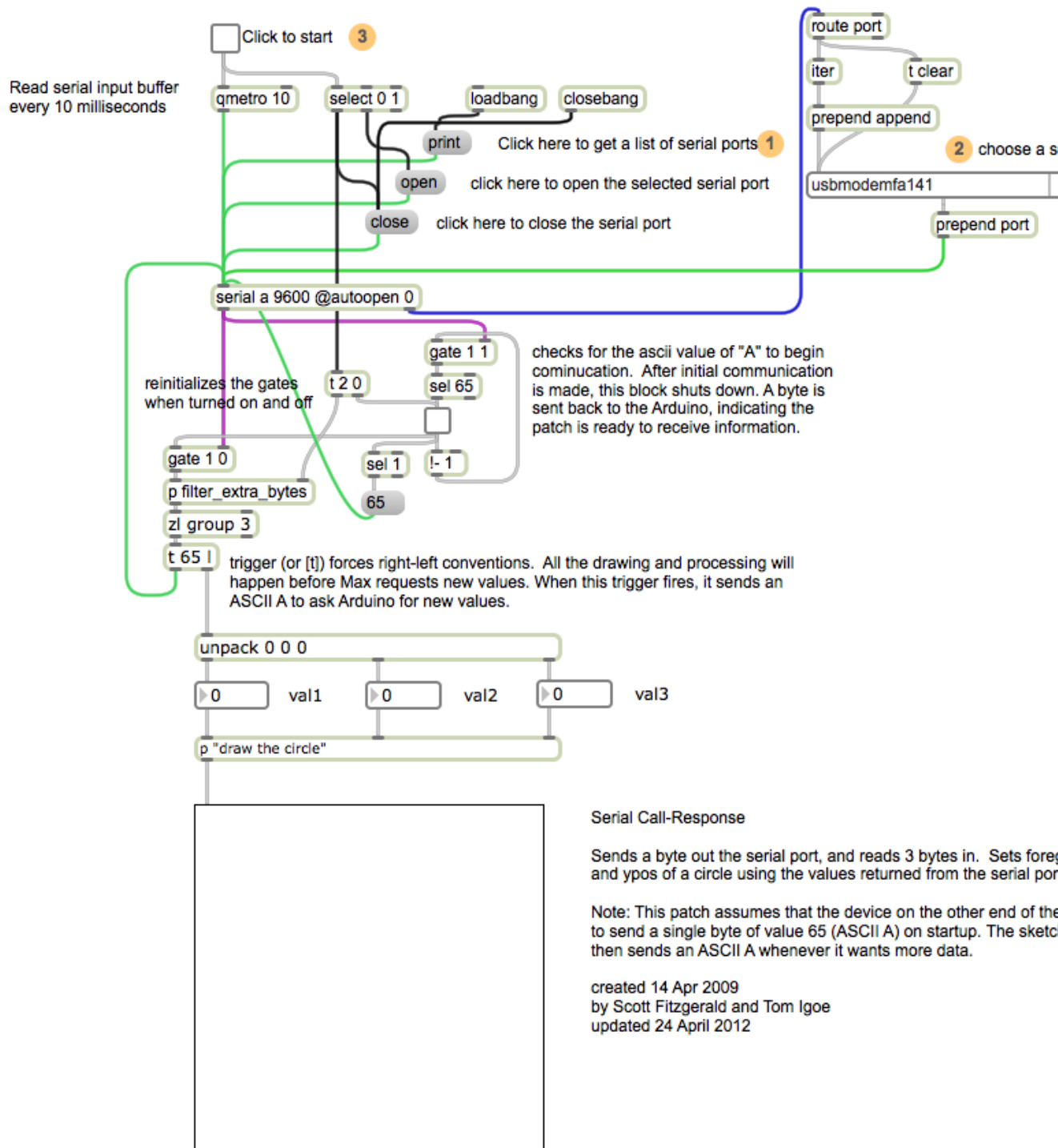
[\[Get Code\]](#)

## Processing Code

Copy the Processing sketch from the code sample above. As you change the value of the analog sensor, you'll get a ball moving onscreen something like this. When you turn the switch off, the ball will disappear:

**Max Code**

The max patch looks like this. Copy the text from the code sample above.



### Serial Call and Response (handshaking) with ASCII-encoded output

This example demonstrates string-based communication from the Arduino board to the computer using a call-and-response (handshaking) method.

The sketch sends an ASCII string on startup and repeats that until it gets a serial response from the computer. Then it sends three sensor values as ASCII-encoded numbers, separated by commas and terminated by a linefeed and carriage return, and waits for another response from the computer.

V1.0

You can use the Arduino serial monitor to view the sent data, or it can be read by Processing (see code below), Flash, PD, Max/MSP (see example below), etc. The examples below split the incoming string on the commas and convert the string into numbers again.

Compare this to the [Serial call and response example](#). They are similar, in that both use a handshaking method, but this one encodes the sensor readings as strings, while the other sends them as binary values. While sending as ASCII-encoded strings takes more bytes, it means you can easily send values larger than 255 for each sensor reading. It's also easier to read in a serial terminal program.

### Hardware Required

Arduino Board

(2) analog sensors (potentiometer, photocell, FSR, etc.)

(1) momentary switch/button

(3) 10K ohm resistors

breadboard

hook-up wire

### Software Required

[Processing](#) or

[Max/MSP version 5](#)

### Circuit

Connect analog sensors to analog input pin 0 and 1 with 10Kohm resistors used as voltage dividers. Connect a pushbutton or switch connected to digital I/O pin 2 with a 10Kohm resistor as a reference to ground.

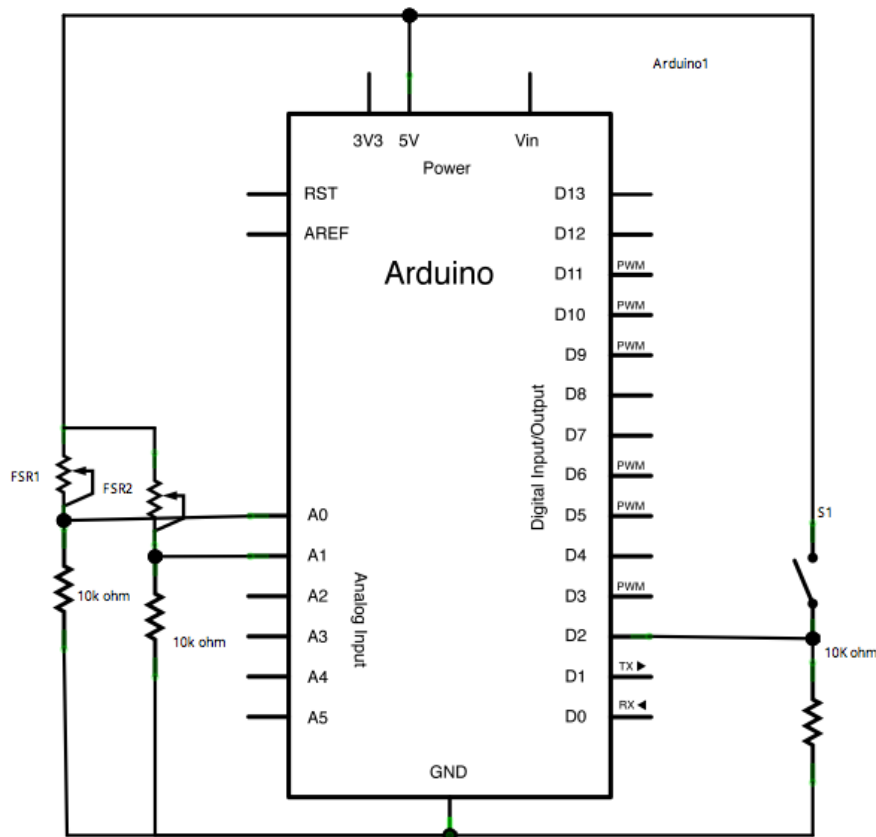
click the image to enlarge



image developed using [Fritzing](#). For more circuit examples, see the [Fritzing project page](#)

### Schematic

click the image to enlarge



## Code

```
/*
```

```
  Serial Call and Response in ASCII
  Language: Wiring/Arduino
```

```
  This program sends an ASCII A (byte of value 65) on startup
  and repeats that until it gets some data in.
  Then it waits for a byte in the serial port, and
  sends three ASCII-encoded, comma-separated sensor values,
  truncated by a linefeed and carriage return,
  whenever it gets a byte in.
```

```
  Thanks to Greg Shakar and Scott Fitzgerald for the improvements
```

```
  The circuit:
```

- \* potentiometers attached to analog inputs 0 and 1
- \* pushbutton attached to digital I/O 2

```
  Created 26 Sept. 2005
```

V1.0

*by Tom Igoe**modified 24 Apr 2012**by Tom Igoe and Scott Fitzgerald**This example code is in the public domain.*<http://www.arduino.cc/en/Tutorial/SerialCallResponseASCII>

```
*/

int firstSensor = 0;    // first analog sensor
int secondSensor = 0;  // second analog sensor
int thirdSensor = 0;   // digital sensor
int inByte = 0;        // incoming serial byte

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

  pinMode(2, INPUT);    // digital sensor is on digital pin 2
  establishContact();   // send a byte to establish contact until
  receiver responds
}

void loop()
{
  // if we get a valid byte, read analog ins:
  if (Serial.available() > 0) {
    // get incoming byte:
    inByte = Serial.read();
    // read first analog input:
    firstSensor = analogRead(A0);
    // read second analog input:
    secondSensor = analogRead(A1);
    // read switch, map it to 0 or 255L
    thirdSensor = map(digitalRead(2), 0, 1, 0, 255);
    // send sensor values:
    Serial.print(firstSensor);
    Serial.print(",");
  }
}
```



V1.0

```

    Serial.print(secondSensor);
    Serial.print(",");
    Serial.println(thirdSensor);
  }
}

void establishContact() {
  while (Serial.available() <= 0) {
    Serial.println("0,0,0"); // send an initial string
    delay(300);
  }
}

/*
Processing code to run with this example:

// This example code is in the public domain.

import processing.serial.*; // import the Processing serial
library
Serial myPort;             // The serial port

float bgcolor;             // Background color
float fgcolor;             // Fill color
float xpos, ypos;         // Starting position of the ball

void setup() {
  size(640,480);

  // List all the available serial ports
  println(Serial.list());

  // I know that the first port in the serial list on my mac
  // is always my Arduino module, so I open Serial.list()[0].
  // Change the 0 to the appropriate number of the serial port
  // that your microcontroller is attached to.
  myPort = new Serial(this, Serial.list()[0], 9600);

  // read bytes into a buffer until you get a linefeed (ASCII 10):
  myPort.bufferUntil('\n');

  // draw with smooth edges:
  smooth();

```

V1.0

```

}

void draw() {
    background(bgcolor);
    fill(fgcolor);
    // Draw the shape
    ellipse(xpos, ypos, 20, 20);
}

// serialEvent method is run automatically by the Processing applet
// whenever the buffer reaches the byte value set in the
bufferUntil()
// method in the setup():

void serialEvent(Serial myPort) {
    // read the serial buffer:
    String myString = myPort.readStringUntil('\n');
    // if you got any bytes other than the linefeed:
    myString = trim(myString);

    // split the string at the commas
    // and convert the sections into integers:
    int sensors[] = int(split(myString, ','));

    // print out the values you got:
    for (int sensorNum = 0; sensorNum < sensors.length; sensorNum++)
    {
        print("Sensor " + sensorNum + ": " + sensors[sensorNum] +
"\t");
    }
    // add a linefeed after all the sensor values are printed:
    println();
    if (sensors.length > 1) {
        xpos = map(sensors[0], 0,1023,0,width);
        ypos = map(sensors[1], 0,1023,0,height);
        fgcolor = sensors[2];
    }
    // send a byte to ask for more data:
    myPort.write("A");
}

*/

/*

```

Max/MSP version 5 patch to run with this example:

-----begin\_max5\_patcher-----

```
3640.3oc6cs0jZajE94Y9UzKkeHoVloTeSHkm1II0VkeHIthSs6C1obIjZ.E
KjJHRhY7jT4+9d5KBj.jTCAXfoV6x.sj5VmyWet127ed6MCFm8EQw.z2f9.5l
a9yau4F0kjW3FS4aFLO3KgIAEpGaPX174hzxAC02qT7kR80mkkUHPAnBQdbP
BZQVdIZRdlbT4r3BDTmkU0YQPY3r3zoeJWDVpe2ttr6cFhvXt7KhyH8W26f9
USkhiTulrw+1czQUszjrzzxf4B0sdP9dqtS5x4woIhREQiWewrkkUW0oViTD
+GpFAST2Qd0+51akeLzRPIU7DPXagIFnH.4653f9WAKKyxVHRQNcfDXliH2w
puvbdWHA1cTPBRKHg4x5mr74EBMINHV1+iFL.8qG.VMWTDDLurs.TBH+zAvP
nTEhvvxun9pbD6FWH38DWH6DWv6ItbX.RKBOJ7XbP5ztvDesvhBLb6VTwcOg
DmiBjnXfiIrjjED0CpP490PEmtPEXwQA5EGUVjK.CKQJqtcYl0nCMRAJi76D
Z7dQf1CCVV1i+ENiTy3AwYaghEA4.KVJx+jHMXbhntJPce03iBpPOPktZqtU
jUoXtw28fkEimmEI1OI.3Q4iMT9wO+iLxc9O7sN28928t6Ve8uMYo.7EUN6t
ePVoUW+6E4hOW7CAgeaVlmeWdlcuWnYLy8mKhhClGDd25F3ce+C2si1Ud42+
bZ3IQJOXg7q96t80e50YvDjqHw7VvkRTXhHHuKEerRwmqfBFsS.g9h.HZN3X
hJf5Qd+xHZHgzc.mrqeYjbn4E84evfIDUjDtjNwD2iRHV6anmGdbmsfKxTTJ
dd93rjtBJ2U42foCwZDqKfYzKkrh4VgYIY4FxVRmN2646f8ck+xw7KrzOlZ
ZYAVfdZgKlaWn29FzA8nfdR2quj.3ejf1BJnKr.Dwpf13cZBm85P0rPj.roB
6fvztPFGkVI0SAPi5NKHmih7E8Ph2e35uOtYN6x6JEQtJVWpV7gRtm2dZy9W
+YMCxLHrEvAknQktDVdY7v82SFosgmSGHO56BRRt6mEEKxRKDnGd+2812h9X
5GSeODOcAJ.M9YHHafjPkyD0GIugn.Ht6bQ.7TTS8DoPtCQCQxWobX+jYPUJ
hPn3zgnx7kogphieFZ2j3TwDgH5dzaUscJ77kEnIY4hoYKglVYzch5KKxJzu
qmgegxl.0MLNGBNDsr.5IUz0iAPZFE.0TtLOEdClQYrAAeORwW+XVo3aP+hb
DHUBCH.mfbEKfGOPYjQhGiCAdNUUBRcQjij4X.u5MZRDzHSyTDQfbcYdHHIM
AzlF1lnoLjKG8UZH5guV1vEka4kKWbOPGPC9YgjNdJHVy+ZJQ1.Cq.FUWQpA
ke.8DbUwi.YEWBUChPyAXCEETFbuhICg9EIRiYnGVjKyt0+io.r+9vrXRz+
Nt7OlJxCRhT35u.X0amlI9X5xEQppQwneJrLarPVU7JkGYWVHz2njevz1UoX
XkoEWOkxDWO9kXYocoTwuzF611zXJyimB3F5qf9nOT9qesryJTJ1EOcV4cIh
IPVWYoOBUMFT1.4sGRRzRT4AOIkRjn8h7LnNJI2mhg6OSk5JZrPJ4i9gfu.R
w+NHLCCpfAMij88n+qTPPMt4UTwj3bAnY.h.aIe.RiAEeF8PdZX3zLkLUS1Z
mcmczah0FH4ZmpLcp.rVbX3d0zalKhSiKAXBZ9BU2zTP3uPobg1LQ.U0.kl+
jcBZj1AMOpzsJYjdz0n53QXsfYrqELKblH7yUFoDfPVXbrwDGXqCjwjviT7a
rXZbpxOvxzXvpOnPH0GlTJMzOg812UZJcdPjxjG7ywIYgeFULaInFDk8jpxZ
apvMA4cv9X.7.vaRRGFACPYHMR0dF2BZC7wEJ2TOKeZnCRD+HzJo.OLWSW6r
qk2wfI6pGf.pdjC4rpfl2YeK8JYloVf93.ocJEvocv9wAcEiMQgBtl.lb0y9
heKnvtGRs+iHOJHM3uaZbN1jDrhED4FfwfLPCEmH8jV.BB0Z+aF.Vkqc4apU
EIb9a5zAcGt5Rf3WdsNJ3R4PXDU0mouHzIca0MWO.KpQjT8oq1SIYqV3mP24
ToxfHpdYOPNqgwoK.W.fxfrNtwsidsBv1T9ociSMu+jfPQqUtk9paFLMONJK
URFMpq7xUuvOXF1HBuN6ndhzfE6nxPXQkKKFGjKQNYHtSptYYVVRyasyBD3
CRiA0YQYrlbgHdptY77E4wZk5UWSOf9yJByyRRZzT5673NtiNrvmhiJmoZq5
fI73wKp5DFrBihhmBNxadsxfoEMuRiIbutfVcM4FWuyr.2bvrlNF5.3U+q9C
sKaa5jkMt70iSd8bC2ZbEFUuAa0DWqYF0tJ91p43649br2nZ2usLGuoxrnQq
6TArNx+1CjRLPpVWf62Kj59ZFRa38Y6D0kRo8AnT8b0g0e4p8+f6.P4sBnaX
```

V1.0

```

TqMmPsOdOcJg+dMtOmdzcgLdIGqjX0J+FAVrmSu.L8fAX19Ky1C.e1.z+IB2
qpeCIUV+.I4fARxQGH0i.9ECVZrhZMTheMCkc4XRMsoCgbef2ZFjaF5MXzaH
n2PQuGymhe0WjdcU47Z1Ukhhb6CwFISy2HNtcvtaNRWdshHNVgHcNMUlopRm4
tJBYYLXfIOUN6GM7eUiFTm8BMbctZQC8atOegDu6oveXrgpeaGnfETvsBJN
6AKuNsT4n+zRVXJtQd+ciEEYKyCq.8ptRTSdBRQrLNcUd5eXcjoa7fyhihZ1
UrNQxBYZo5g.vpdt8klkJi1QyPvdH7UFMStbvYu8Amu1nY7ECMKGBqnY2KH
Z18Jj14aYNnEYiQWVzrUxytWNzL0VZ14xglI6isN5kAMi2GZlbYPyNma6FqC
aJR9qEogO+ovfvYFxxjGV07cLnH3QQzm.R.BG7SAkk4wiWVpC2p9jwX23ka
0zSz4M6e1QZY.8mljMNHwLURqZ9FuzslMk8ZJXtcMPeblVut1XYDhdMCpmjZ
8BAqsU9DezKxJAa8Hmbbfi+wccuVv7c0qELrEHB+UAhHWzCfCbKPEyBki24Z
clythVwfkYSmlHrPdX8tC5vliPb5ArPuOWc8NVrRZspq24UxhE0wBcAsMyt2
2LLuqvKkZRXjEq5CM6S3tq9Zm6HD+8Prm0F+jDwnlpaUe+2ZuF259kxkiR5W
Qf6vzKBtMm+gFrMeuWsKW.6B61VyWOFjz0Zsmwza+.ikxQcAL3iDtbLWMTKm
OtyMEFcjWM9iu0rMa81D8kUl3v2ewcHWP5B2HX6kK7t7DL5fs6JVIr00Z113
bEpOP3zih9.gbspPzKDYbRVAQ7CFhtZsYzhW1ko0WEJcG3oAC0aRIyxKsUEI
+iDPwOLfp0uNA6MmtSUSmRuNb8d1ttWya7sVWf5Iwf.1LQtZUnqNvT1bS6z
E5o2vfqNSH5bufQbuZV09M.E04Mj8XBUIBqNGl5FSt3NGlZaGRpV6wc4kiWi
q0twaaoRHull1jjsIi7cMjQlJJUaQuhR495nlfRQWRJXkrgmMGXWjKM4jdGJH
yovkl4HUetutzWuY5tjFHneGn77rtG3iJ92whCVJxKhBwgGtRaFIzabfNrRn
WThd9q24vsZjf9JvHwOKBhprFDmtXYIZ7xISja01GE4OK2V9yiS.qFhvrznh
8cKyMZs7EVEpT01FlCe0rIC01Uk6NX4N9syCyAE660+ovE9hyGqjaGurrLak
G0YwoMlFO4YMSZjd9DcWucsJUr1Yqgy8TluCY3N9Q8.+k0JCD3ZTS0CW8Qyb
s19nOxrgjw7VFU+3ooYviK66pCfimt8AAxHOOBkK+EajC2yayWtciMzgdpM
NKORj29YyGcS4wFVlql0wcZTg1yw5wvMNItpuUzpu.Y0miRlg00w7wpZI2Em
SUBGayVM5eqU4C+rV4ZSPkvXqLJbAHLR3mKwT5ISL8+Kv0k.GWEKwpP3ewk3
7omKIN7EtDmp4ZtHk0BfatXgLhgashgZrVYaY8AIO7fq8Pas1fFzjd4ibwpd
XO4GXOeOG+lcyasNh1R+wVx2yBxeTOT+wiZFYA0P48PNYiiVjAhJlNT4Qvpb
uj3aN2qYqJcBfSWhMbf+YCPcsfbNeTC219WNC+5eIlkST0RJgupzIn+kysgC
X6GGXnYpdYfP0GP6MKQXM3N1Ih6XVvcLuym7B0B5w8v.ahqBI49qJcJ.TaX.
N+xBP4NGHhhqYfkrNM9q1f3ZweqyYCQYdGCSZGQ5wBx47o.Ssw+CkcgQOmod
KZic4QKzCw+7ROM8nY2LfMsEDtdfeMKS5Ev95IQhorcqJcBrzPsQUhRNe8M
1X6lhOezC4BidvlnKcFs8YimJ9n8RWZXi07aSCxDRLdjd91qU5TnmXCeRvmR
9jnm7b15RmJ9rO4Kr+IgO04BfczyOpqx9npzofOsIlaR8Mo0IUMR48i0mYly
lVMw1w6gbloGRezy4yKEw6BHBBWik.eRi3DNM5KDahS.SOE1EjmXl7Uyqo9T
AtQAO8fG3oLX3cZFxKh0FLNSRfDaoG74gdvW.ZDU9FMGSdFMBt+IQh.6eIvw
FujTkJREGKKcJ3X2WtXf7Ub1HywEqxh2tJnE.FcZhMByrcXQw1x+bOWJYjpy
lv8oq55aEHLcwD8hJjxbVU5EigcNtL7Ql76KVvp69Huhcb87vpOckRYT+96v
Hd5AylrofMqm+FkLYvv0+GL3FkL6bLp21kL6QFNV8BNM48foWBV4zt1wXm5V
4jkNEbL45dtNw13Iltmi9sAyY0S018BR+3yWjVXax7eOmKrp4m0QKIal6VYo
SAf5XQxSrCa510qk45k5kAzqEgMNgzkz9FmL5abpnu4IhNzZ+0s+OKCSg0.
-----end_max5_patcher-----

```

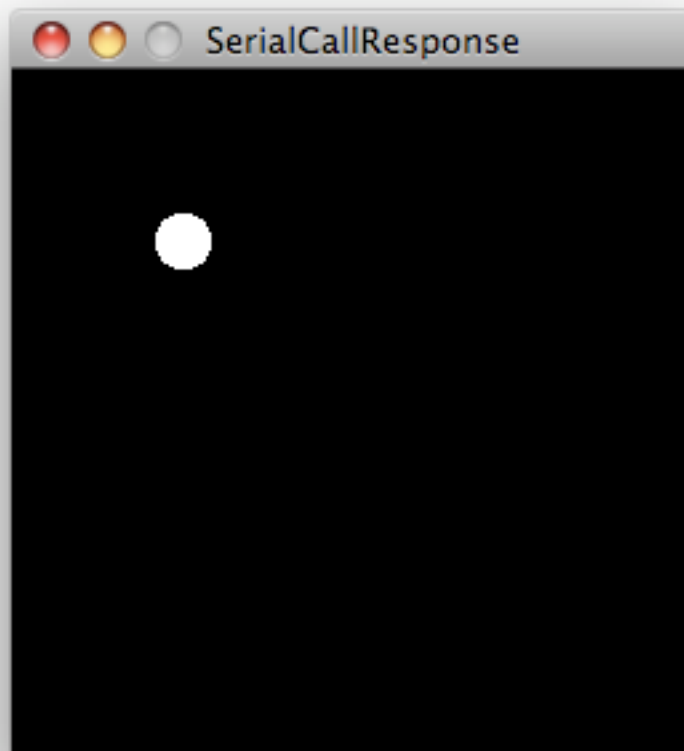
\*/

[\[Get Code\]](#)

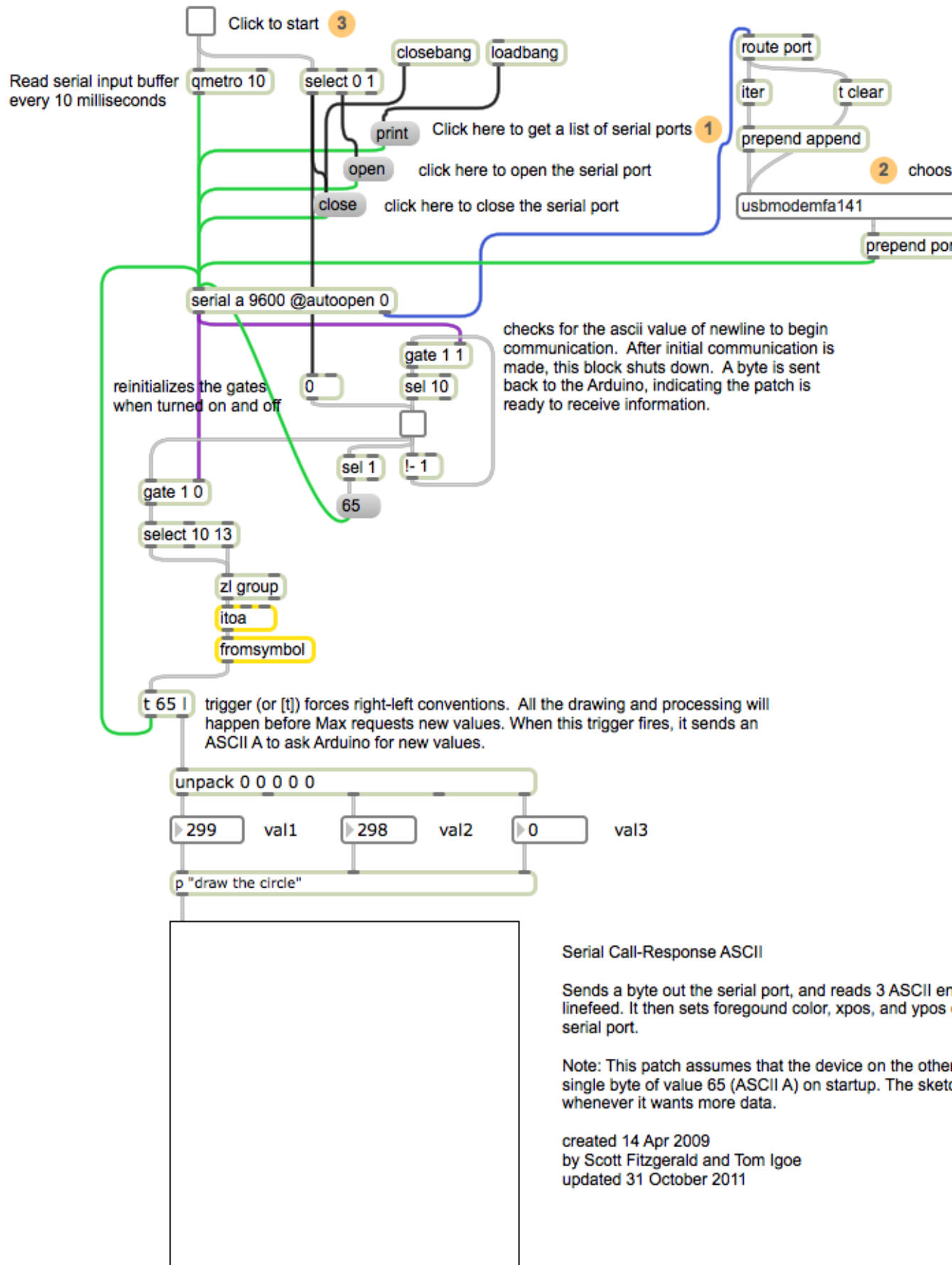
V1.0

**Processing Code**

Copy the Processing sketch from the code sample above. As you change the value of the analog sensor, you'll get a ball moving onscreen something like this. When you turn the switch off, the ball will disappear:

**Max Code**

The max patch looks like this. Copy the text from the code sample above and paste it into a new Max window



## 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 :

Serial

Wire

Ethernet Client

Ethernet Server

SD

## Functions

available()

read()

flush()

find()

findUntil()

peek()

readBytes()

readBytesUntil()

parseInt()

parseFloat()

setTimeout()

## 1 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

### See also

[Stream](#)

## 2 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)

### See also

[Stream](#)

## 3 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



**See also****4 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

**See also**

[Stream](#)

[Reference H](#)

**5 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

**6 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.

V1.0

**Returns**

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

**See Also**

## 7 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 characters 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**

byte

## 8 readBytesUntil()

**Description**

readBytesUntil() read 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 characters 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**

byte

**See also**

[Stream](#)

[Reference Home](#)

## 9 parseInt()

### Description

parseInt() returns the first valid (long) integer number from the current position. Initial characters that are not integers (or the minus sign) are skipped. parseInt() is terminated by the first character that is not a digit.

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*)

### Parameters

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

*list* : the stream to check for ints (char)

### Returns

int

### See

## 10 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

### See

## 11 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).

V1.0

---

**Parameters**

None

SMRAZA