STI2D	Language Arduino	

Structure

setup()

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

Example

```
int buttonPin = 3;

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

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

loop()

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

Example

```
const int buttonPin = 3;

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

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

  delay(1000);
}
```

Control Structures

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

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

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

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

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

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

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

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

if (x > 120){
  digitalWrite(LEDpin1, HIGH);
  digitalWrite(LEDpin2, HIGH);
}                              // all are correct
```

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

Comparison Operators:

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

Warning:

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

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

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

if / else

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

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

else can proceed another **if** test, so that multiple, mutually exclusive tests can be run at the same time.

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

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

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

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

for statements

Description

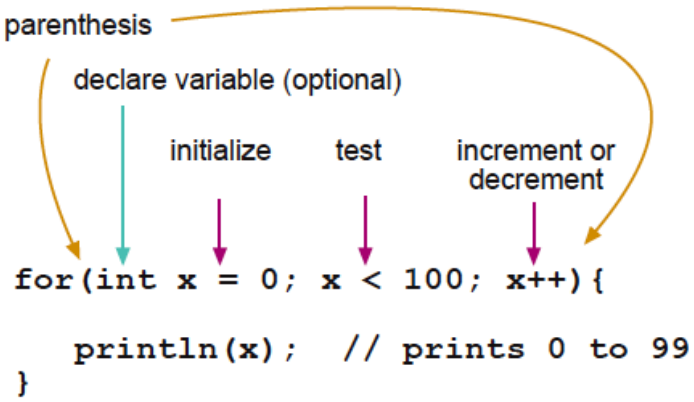
The **for** statement is used to repeat a block of statements enclosed in curly braces. An increment counter is usually used to increment and terminate the loop. The **for** statement is useful for any repetitive operation, and is often used in combination with arrays to operate on collections of data/pins.

There are three parts to the **for** loop header:

for (**initialization**; **condition**; **increment**) {

//statement(s);

}



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);
  }
}
```

See also

- while

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
     // 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
See also:

if...else Reference Home

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
{
   // 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.

Example

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

Further Syntax

; semicolon

Used to end a statement.

Example

```
int a = 13;
```

Tip

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

{} Curly Braces

Curly braces (also referred to as just "braces" or as "curly brackets") are a major part of the C programming language. They are used in several different constructs, outlined below, and this can sometimes be confusing for beginners.

An opening curly brace "{" must always be followed by a closing curly brace "}". This is a condition that is often referred to as the braces being balanced. The Arduino IDE (integrated development environment) includes a convenient feature to check the balance of curly braces. Just select a brace, or even click the insertion point immediately following a brace, and its logical companion will be highlighted.

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

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

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

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

The main uses of curly braces

Functions

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

Loops

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

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

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

Conditional statements

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

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

Single line comment

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.

Multi line comment

Comments are lines in the program that are used to inform yourself or others about the way the program works. They are ignored by the compiler, and not exported to the processor, so they don't take up any space on the Atmega chip.

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

Example

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

/* this is multiline comment - use it to comment out whole blocks of code

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

Tip

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

#Define

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

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

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

Arduino defines have the same syntax as C defines:

Syntax

#define constantName value

Note that the # is necessary.

Example

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

Tip

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

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

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

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

#include

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

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

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

Example

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

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

prog_uint16_t myConstants[] PROGMEM = {0, 21140, 702  , 9128,  0, 25764, 8456,
0,0,0,0,0,0,0,0,29810,8968,29762,29762,4500};
```

Arithmetic Operators

= assignment operator (single equal sign)

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

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

Example

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

Programming Tips

The variable on the left side of the assignment operator (= sign) needs to be able to hold the value stored in it. If it is not large enough to hold a value, the value stored in the variable will be incorrect.

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

Addition, Subtraction, Multiplication, & Division

Description

These operators return the sum, difference, product, or quotient (respectively) of the two operands. The operation is conducted using the data type of the operands, so, for example, 9 / 4 gives 2 since 9 and 4 are ints. This also means that the operation can overflow if the result is larger than that which can be stored in

the data type (e.g. adding 1 to an int with the value 32,767 gives -32,768). If the operands are of different types, the "larger" type is used for the calculation.

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

Examples

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

Syntax

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

Parameters:

value1: any variable or constant

value2: any variable or constant

Programming Tips:

- Know that integer constants default to int, so some constant calculations may overflow (e.g. 60 * 1000 will yield a negative result).
- Choose variable sizes that are large enough to hold the largest results from your calculations
- Know at what point your variable will "roll over" and also what happens in the other direction e.g. (0 - 1) OR (0 - - 32768)
- For math that requires fractions, use float variables, but be aware of their drawbacks: large size, slow computation speeds
- Use the cast operator e.g. (int)myFloat to convert one variable type to another on the fly.

% (modulo)

Description

Calculates the remainder when one integer is divided by another. It is useful for keeping a variable within a particular range (e.g. the size of an array).

Syntax

result = dividend % divisor

Parameters

dividend: the number to be divided

divisor: the number to divide by

Returns

the remainder

Examples

```
x = 7 % 5;   // x now contains 2
x = 9 % 5;   // x now contains 4
x = 5 % 5;   // x now contains 0
x = 4 % 5;   // x now contains 4
```

Example Code

```
/* update one value in an array each time through a loop */

int values[10];
int i = 0;

void setup() {}

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

Tip

The modulo operator does not work on floats.

Boolean Operators

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

&& (logical and)

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

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

is true only if both inputs are high.

|| (logical or)

True if either operand is true, e.g.

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

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

! (not)

True if the operand is false, e.g.

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

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

Warning

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

Similarly, do not confuse the boolean || (double pipe) operator with the bitwise OR operator | (single pipe).

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

Examples

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

See also

- & (bitwise AND)
- | (bitwise OR)
- ~ (bitwise NOT
- if

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 and knowledge of manipulating pointers is handy to have in one's toolkit.

Bitwise Operators

Bitwise AND (&), Bitwise OR (|), Bitwise XOR (^)

Bitwise AND (&)

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

Description and Syntax

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

Bitwise AND (&)

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

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

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

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

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

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

Bitwise OR (|)

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

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

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

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

Example Program

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 biwise OR).

```
int i;    // counter variable
int j;

void setup(){
DDRD = DDRD | B11111100; // set direction bits for pins 2 to 7, leave 0 and 1 untouched (xx | 00 == xx)
// same as pinMode(pin, OUTPUT) for pins 2 to 7
Serial.begin(9600);
}

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

PORTD = PORTD & B00000011;  // clear out bits 2 - 7, leave pins 0 and 1 untouched (xx & 11 == xx)
j = (i << 2);               // shift variable up to pins 2 - 7 - to avoid pins 0 and 1
PORTD = PORTD | j;          // combine the port information with the new information for LED pins
Serial.println(PORTD, BIN); // debug to show masking
delay(100);
  }
}
```

Bitwise XOR (^)

There is a somewhat unusual operator in C++ called bitwise EXCLUSIVE OR, also known as bitwise XOR. (In English this is usually pronounced "eks-or".) The bitwise XOR operator is written using the caret symbol ^. This operator is very similar to the bitwise OR operator |, only it evaluates to 0 for a given bit position when both of the input bits for that position are 1:

```
0  0  1  1    operand1
```

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

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

Here is a simple code example:

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

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

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

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

Bitwise NOT (~)

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

```
  0  1    operand1

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

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

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

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

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

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

Description

From *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 << number_of_bits

variable >> number_of_bits

Parameters

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

Example:

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

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

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

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

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

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

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

This behavior, called sign extension, is often not the behavior you want. Instead, you may wish zeros to be

shifted in from the left. It turns out that the right shift rules are different for unsigned int expressions, so you can use a typecast to suppress ones being copied from the left:

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

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

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

Compound Operators

++ (increment) / -- (decrement)

Description

Increment or decrement a variable

Syntax

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

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

Parameters

x: an integer or long (possibly unsigned)

Returns

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

Examples

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

+= , -= , *= , /=

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;
```

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;
```

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 | B00000000 = myByte;

Bits that are "bitwise ORed" with 1 are set to 1 so:
myByte | B11111111 = B11111111;

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

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.

LED_BUILTIN

Most Arduino boards have a pin connected to an on-board LED in series with a resistor. LED_BUILTIN is a drop-in replacement for manually declaring this pin as a variable. Most boards have this LED connected to digital pin 13.

Integer Constants

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

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

Base	Example	Formatter	Comment
10 (decimal)	123	none	
2 (binary)	B1111011	leading 'B'	only works with 8 bit values (0 to 255) characters 0-1 valid
8 (octal)	0173	leading "0"	characters 0-7 valid

16 (hexadecimal) 0x7B leading "0x" characters 0-9, A-F, a-f valid

Decimal is base 10. This is the common-sense math with which you are acquainted. Constants without other prefixes are assumed to be in decimal format.

Example:

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

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

Example:

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

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

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

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

Example:

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

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

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

Example:

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

U & L formatters

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

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

floating point constants

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

Examples:

n = .005;

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

floating-point constant	evaluates to:	also evaluates to:
10.0	10	
2.34E5	2.34 * 10^5	234000
67e-12	67.0 * 10^-12	.000000000067

Data Types

void

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

Example:

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

void setup()
{
  // ...
}

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

boolean

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

Example

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

boolean running = false;
```

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

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

char

Description

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

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

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

Example

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

Example

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

int

Description

Integers are your primary data-type for number storage.

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

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

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

Example

```
int ledPin = 13;
```

Syntax

```
int var = val;
```

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

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. Example for a 16-bit int:

```
int x;
x = -32768;
x = x - 1;      // x now contains 32,767 - rolls over in neg. direction

x = 32767;
x = x + 1;      // x now contains -32,768 - rolls over
```

unsigned int

Description

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

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

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

Example

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

Syntax

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

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

Coding Tip

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

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

word

Description

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.

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

Example

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

Syntax

```
long var = val;
```

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

unsigned long

Description

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

Example

```
unsigned long time;

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

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

Syntax

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

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

short

Description

A short is a 16-bit data-type.

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

Example

```
short ledPin = 13;
```

Syntax

```
short var = val;
```

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

float

Description

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

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

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

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

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

Examples

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

Syntax

```
float var = val;
```

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

Example Code

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

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

double

Description

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

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

String (char array)

Description

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

String (object)

Description

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

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

Arrays

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

Creating (Declaring) an Array

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

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

You can declare an array without initializing it as in myInts.

In myPins we declare an array without explicitly choosing a size. The compiler counts the elements and creates an array of the appropriate size.

Finally you can both initialize and size your array, as in mySensVals. Note that when declaring an array of type char, one more element than your initialization is required, to hold the required null character.

Accessing an Array

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

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

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

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

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

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

To assign a value to an array:

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

To retrieve a value from an array:

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

Arrays and FOR Loops

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

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

Example

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

Conversion

char()

Description

Converts a value to the char data type.

Syntax

char(x)

Parameters

x: a value of any type

Returns

char

byte()

Description

Converts a value to the byte data type.

Syntax

byte(x)

Parameters

x: a value of any type

Returns

byte

int()

Description

Converts a value to the int data type.

Syntax

int(x)

Parameters

x: a value of any type

Returns

int

word()

Description

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

Syntax

word(x)
word(h, l)

Parameters

x: a value of any type

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

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

Returns

word

long()

Description

Converts a value to the long data type.

Syntax

long(x)

Parameters

x: a value of any type

Returns

long

float()

Description

Converts a value to the float data type.

Syntax

float(x)

Parameters

x: a value of any type

Returns

float

Variable Scope & Qualifiers

Variable Scope

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

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

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

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

Example:

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

void setup()
{
  // ...
}

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

  for (int j = 0; j <100; j++){
```

```
  // variable j can only be accessed inside the for-loop brackets
  }

}
```

Static

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

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

Example

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

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

int thisTime;
int total;

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

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

int randomWalk(int moveSize){
  static int  place;     // variable to store value in random walk - declared static so that it stores
                         // values in between function calls, but no other functions can change its value

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

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

  return place;
}
```

volatile keyword

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

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

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

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

Example

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

int pin = 13;
volatile int state = LOW;

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

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

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

const keyword

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

Constants defined with the *const* keyword obey the rules of *variable scoping* that govern other variables.

This, and the pitfalls of using*#define*, makes the *const* keyword a superior method for defining constants and is preferred over using *#define*.

Example

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

// ....

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

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

#define or const

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

Utilities

sizeof

Description

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

Syntax

sizeof(variable)

Parameters

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

Example code

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

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

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

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

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

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

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

Functions

Digital I/O

pinMode()

Description

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

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

Syntax

pinMode(pin, mode)

Parameters

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

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

Returns

None

Example

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

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

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

Note

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

digitalWrite()

Description

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

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

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

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

Syntax

digitalWrite(pin, value)

Parameters

pin: the pin number

value: HIGH or LOW

Returns

none

Example

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

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

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

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

Note

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

digitalRead()

Description

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

Syntax

digitalRead(pin)

Parameters

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

Returns

HIGH or LOW

Example

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

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

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

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

Note

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

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

Analog I/O

analogReference(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.

Warning

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

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

analogRead()

Description

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

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

Syntax

analogRead(pin)

Parameters

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

Returns

int (0 to 1023)

Note

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

Example

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

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

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

analogWrite()

Description

Writes an analog value (PWM wave) to a pin. Can be used to light a LED at varying brightnesses or drive a motor at various speeds. After a call to **analogWrite()**, the pin will generate a steady square wave of the specified duty cycle until the next call to **analogWrite()** (or a call to **digitalRead()** or **digitalWrite()** on the same pin). The frequency of the PWM signal on most pins is approximately 490 Hz. On the Uno and similar boards, pins 5 and 6 have a frequency of approximately 980 Hz. Pins 3 and 11 on the Leonardo also run at 980 Hz.

On most Arduino boards (those with the ATmega168 or ATmega328), this function works on pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega, it works on pins 2 - 13 and 44 - 46. Older Arduino boards with an ATmega8 only support analogWrite() on pins 9, 10, and 11.

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

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

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

Syntax

analogWrite(pin, value)

Parameters

pin: the pin to write to.

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

Returns

nothing

Notes and Known Issues

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

Example

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

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

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

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

Due only

analogReadResolution()

Description

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

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

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

Syntax

analogReadResolution(bits)

Parameters

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

Returns

None.

Note

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

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

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

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

Example

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

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

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

```
  Serial.print(analogRead(A0));

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

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

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

analogWriteResolution()

Description

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

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

The Due has the following hardare capabilities:

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

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

Syntax

analogWriteResolution(bits)

Parameters

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

Returns

None.

Note

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

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

Example

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

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

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

  // change the PWM resolution to 12 bits
  // the full 12 bit resolution is only supported
  // on the Due
  analogWriteResolution(12);
  analogWrite(12, map(sensorVal, 0, 1023, 0, 4095));
  Serial.print(" , 12-bit PWM value : ");
  Serial.print(map(sensorVal, 0, 1023, 0, 4095));

  // change the PWM resolution to 4 bits
  analogWriteResolution(4);
  analogWrite(13, map(sensorVal, 0, 1023, 0, 127));
```

```
  Serial.print(", 4-bit PWM value : ");
  Serial.println(map(sensorVal, 0, 1023, 0, 127));

  delay(5);
}
```

Advanced I/O

tone()

Description

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

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

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

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

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

Syntax

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

Parameters

pin: the pin on which to generate the tone

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

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

Returns

nothing

noTone()

Description

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

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

Syntax

noTone(pin)

Parameters

pin: the pin on which to stop generating the tone

Returns

nothing

shiftOut()

Description

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

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

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

Syntax

shiftOut(dataPin, clockPin, bitOrder, value)

Parameters

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

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

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

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

Returns

None

Note

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

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

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

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

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

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

```
}

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

shiftIn()

Description

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

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

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

Syntax

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

Parameters

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

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

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

Returns

the value read (*byte*)

pulseIn()

Description

Reads a pulse (either HIGH or LOW) on a pin. For example, if **value** is **HIGH**, **pulseIn()** waits for the pin to go **HIGH**, starts timing, then waits for the pin to go **LOW** and stops timing. Returns the length of the pulse in microseconds. 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 miliseconds) specified as parameter. (There are 1000 milliseconds in a second.)

Syntax

delay(ms)

Parameters

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

Returns

nothing

Example

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

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

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

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

Caveat

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

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

delayMicroseconds()

Description

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

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

Syntax

delayMicroseconds(us)

Parameters

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

Returns

None

Example

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

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

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

```
  delayMicroseconds(50);        // pauses for 50 microseconds
}
```

configures pin number 8 to work as an output pin. It sends a train of pulses 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(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(senVal, 20); // assigns sensVal to the larger of sensVal or 20
                           // (effectively ensuring that it is at least 20)
```

Note

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

Warning

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

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

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*)

sqrt(x)

Description

Calculates the square root of a number.

Parameters

x: the number, any data type

Returns

double, the number's square root.

Trigonometry

sin(rad)

Description

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

Parameters

rad: the angle in radians (*float*)

Returns

the sine of the angle (*double*)

cos(rad)

Description

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

Parameters

rad: the angle in radians (*float*)

Returns

The cos of the angle ("double")

tan(rad)

Description

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

Parameters

rad: the angle in radians (*float*)

Returns

The tangent of the angle (*double*)

Random Numbers

randomSeed(seed)

Description

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

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

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

Parameters

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

Returns

no returns

Example

```
long randNumber;

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

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

  delay(50);
}
```

random()

Description

The random function generates pseudo-random numbers.

Syntax

random(max)
random(min, max)

Parameters

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

max - upper bound of the random value, exclusive

Returns

a random number between min and max-1 (*long*)

Note:

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

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

Example

```
long randNumber;
```

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

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

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

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

  delay(50);
}
```

Bits and Bytes

lowByte()

Description

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

Syntax

lowByte(x)

Parameters

x: a value of any type

Returns

byte

highByte()

Description

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

Syntax

highByte(x)

Parameters

x: a value of any type

Returns

byte

bitRead()

Description

Reads a bit of a number.

Syntax

bitRead(x, n)

Parameters

x: the number from which to read

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

Returns

the value of the bit (0 or 1).

bitWrite()

Description

Writes a bit of a numeric variable.

Syntax

bitWrite(x, n, b)

Parameters

x: the numeric variable to which to write

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

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

Returns

none

bitSet()

Description

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

Syntax

bitSet(x, n)

Parameters

x: the numeric variable whose bit to set

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

Returns

none

bitClear()

Description

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

Syntax

bitClear(x, n)

Parameters

x: the numeric variable whose bit to clear

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

Returns

none

bit()

Description

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

Syntax

bit(n)

Parameters

n: the bit whose value to compute

Returns

the value of the bit

External Interrupts

attachInterrupt()

Description

Specifies a named Interrupt Service Routine (ISR) to call when an 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 table below shows the available interrupt pins on various boards.

Board	int.0	int.1	int.2	int.3	int.4	int.5
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
Leonardo	3	2	0	1	7	
Due	**(see below)**					

The Arduino Due board has powerful interrupt capabilities that allows you to attach an interrupt function on all available pins. You can directly specify the pin number in attachInterrupt().

Note

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

Using Interrupts

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

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

About Interrupt Service Routines

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

Generally, an ISR should be as short and fast as possible. If your sketch uses multiple ISRs, only one can run at a time, other interrupts will be ignored (turned off) until the current one is finished. as delay() and millis() both rely on interrupts, they will not work while an ISR is running. delayMicroseconds(), which does not rely on interrupts, will work as expected.

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

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

Syntax

attachInterrupt(interrupt, ISR, mode)

attachInterrupt(pin, ISR, mode) (*Arduino Due only*)

Parameters

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

pin: the pin number (*Arduino Due only*)

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

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

- **LOW** to trigger the interrupt whenever the pin is low,
- **CHANGE** to trigger the interrupt whenever the pin changes value
- **RISING** to trigger when the pin goes from low to high,
- **FALLING** for when the pin goes from high to low.

The Due board allows also:

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

Returns

none

Example

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

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

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

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

detachInterrupt()

Description

Turns off the given interrupt.

Syntax

detachInterrupt(*interrupt*)

detachInterrupt(*pin*) (*Arduino Due only*)

Parameters

- *interrupt*: the number of the interrupt to disable (see attachInterrupt() for more details).
- *pin*: the pin number of the interrupt to disable (*Arduino Due only*)

Interrupts

interrupts()

Description

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

Parameters

None

Returns

None

Example

```
void setup() {}

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

noInterrupts()

Description

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

Parameters : **None.**

Returns

None.

Example

```
void setup() {}

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

Communication

Serial

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

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

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

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

The Arduino Leonardo board uses **Serial1** to communicate via TTL (5V) serial on pins 0 (RX) and 1 (TX). **Serial** is reserved for USB CDC communication. For more information, refer to the Leonardo getting started page and hardware page.

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