Comments

```
// single line

/* multiple
line */
```

Syntax

```
{}      encapsulate code
;       line ending
```

Variable Declaration

```
#define constantName 42
const type constantName;

//forced unsigned long
const int constantName = 32767ul;

type variableName;
type variableName = value;

// can also use 1 and 0
boolean variableName = false;

// for letters only, see also "string"
// SIGNED byte, -128 to 127
char variableName = 'A'; //equivalent.
char variableName = 65;

// 0 to 255
// binary number declaration
byte variableName = B10010;

// -32,768 to 32,767
// i.e. -2^15 to (2^15) - 1)
// hexadecimal declaration shown
int variableName = 0x7B;

//0 to 65,535
//i.e. (2^16) - 1)
unsigned int variableName = 42000;
word variableName = 42000;

// -2,147,483,648 to 2,147,483,647
// i.e. -2^32 to (2^31) - 1)
long variableName = i++;

//0 to 4,294,967,295
//i.e. (2^32 - 1)
//shows returning function
unsigned long variableName = millis();

//3.4028235E+38 and as low as
//-3.4028235E+38
//(32 bit but with only 6-7 decimal
//places of precision for _both_ floats
//and doubles)
float variableName = 3.1459;
double variableName = 3.1459;
```

Arrays:

```
//arrays are 0 indexed.

// will be an array of 6 items
const int myArrayLength = 6;
type myArray[myArrayLength];

// an array 6 long, all positions full
type myArray[] = {2, 4, 8, 3, 6, 9};

// will be an array 6 long,
// positions 5 and 6 will be empty
const int myArrayLength = 6;
type myArray[myArrayLength] =  {2, 4, 6, 9};

some standard uses:
int i;
for (i = 0; i < myArrayLength; i = i + 1) {
  Serial.println(myArray[i]);
}

BONUS: (google pointers in C )
int *ptr;
ptr = &my_array[0];
```

Function Declarations

```
void myFunction(){
  //do something
}

//function that returns it's own parameter.
//in this case types must match!
type myFunction(type myParameterName){
 type returnValue = myParameterName;
 return returnValaue;
}

void myFunction(byte * someArray) {}
```

Basics Operators

```
Comparison Operators

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

Boolean Operators

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

Bitwise Operators

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

Compound Operators

++ (increment)
-- (decrement)
+= (compound addition)
-= (compound subtraction)
*= (compound multiplication)
/= (compound division)

&= (compound bitwise and)
|= (compound bitwise or)
```

Control Structures

if, else and if else

```
if (x >= 120 || x <= 30) digitalWrite(LEDpin, HIGH);

if (x > 120 && y != 6)
digitalWrite(LEDpin, HIGH);

if (!x){ digitalWrite(LEDpin, HIGH); }

if (x > 120){
  digitalWrite(LEDpin1, HIGH);
  digitalWrite(LEDpin2, HIGH);
}


if (boolean test condition)
{
}
else if (other boolean test condition)
{
}
else //default to...
{
}
```

for loops

```
for (int i=startValue; i <= endValue; i++){
// statement(s)
}

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

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

while and do while

```
while(boolean test condition){
  // statement(s)

  //then if you need to bail out
  if (some other test condition){
     break;
  }
}

do
{
 // statement block always runs tales once
} while (boolean test condition);
```

case statement

```
switch (var) {
   case 1:
      //do something when var equals 1
      break;
    case 2:
      //do something when var equals 2
      break;
    case 86:
      //do something when var equals 86
      // you can jump around!
      break;
    case someConstantName:
      //do something when var equals
      // a constant defined at the top
      // of your code
      // you can jump around!
      break;
    default:
      // if nothing else matches, do the default
      // default is optional
  }
```

Digital I/O Examples

pinMode()

```
 for (byte i = 0; i <= myPinArrayLength; i ++) {
    pinMode(pinArray[i], OUTPUT);
  }

for (byte i = 0; i <= mySwtchAryLength; i ++) {
    pinMode(switchArray[i], INPUT);

    // for high impedance usage…
    //(looking for 0 not for 1)
    digitalWrite(switchArray[i], HIGH);
  }
```

digitalWrite()

```
digitalWrite(ledPin, HIGH); //true, 1
delay(1000);
digitalWrite(ledPin, LOW); //false, 0
delay(1000);
```

non blocking toggle snippet:

```
void blinkIt(int myLED, int myBlinkPeriod) {
  if
  ((myBlinkPeriod) < (currentMillis- blinkFlipTime)) {
    blinkState ? blinkState=false : blinkState=true;
    blinkFlipTime = currentMillis;
  }
    digitalWrite(myLED,blinkState);
}
```

digitalRead()

```
variable = digitalRead(inPin);
```

dependency snippet:

```
void pickLED() {
  int toggleButtonState;
  toggleButtonState = digitalRead(toggleButtonPin);
  if (toggleButtonState == HIGH) {
    currentLED = ledPinOne;
    otherLED = ledPinTwo;
  }
  else {
    currentLED = ledPinTwo;
    otherLED = ledPinOne;
  }
}
```

Analog I/O Examples

analogRead()

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

analogWrite()

```
//must be on one of the PWM Pins
//9,10,11 NEW-> 3,5,6
//Must be a value 0-255
analogWrite(ledPin, 255);
```

map()

```
//linear mapping (i.e. normalization function)
blinkOnPeriod = map(sensorValue, sensorMin,
sensorMax , blinkShortest , blinkLongest);

//non variable pimped out
byte myPWM = map(sensorValue, 0, 1023, 0, 255);
```

old way was something like:
(newMax - newMin) / (oldMax-oldMin) * valueToBeMaped

constrain()

```
//truncates values to fit
int prntblChar = constrain(inByte,32,126);
```

Serial Sending

Serial.begin()

common rates & size variable it would take to hold them:

int	300
int	1200
int	4800
int	9600
int	14400
int	19200
int	28800
word	38400
word	57600
long	115200

```
int baudrate = 9600;

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

there is a Serial.end but it is uncommon, especially wen the begin is only in the setup!

Serial.print()

```
//how each of these would handle
someValue = 65;

//depending on what you send it to
//might give you a "A"
Serial.print(someValue, BYTE);

//ASCII encoded binary "1000001"
Serial.print(someValue, BIN);

//ASCII encoded decimal "65"
Serial.print(someValue, DEC);

//ASCII encoded hexadecimal "41"
Serial.print(someValue, HEX);

//ASCII encoded octal notation "101"
Serial.print(someValue, OCT);

// print a tab, ASCII 9
Serial.print('\t');

//print a line feed, ASCII 10
Serial.print('\n');

//print a carriage return, ASCII 13
Serial.print('\r');
//more common to just use…
```

Serial.println()

```
someValue = 65;

//prints a 65 followed by a
Serial.println(someValue, DEC);
```

Serial.write()

```
someValue = 65;

//depending on what you send it to
//might render as "A" but it is just the
//idea of 65, less than a byte of
//information, vs "65" which is two
//bytes
Serial.write(someValue);
```

Serial Receiving:

Serial.available()

```
//if there is nothing waiting for me to read…
void establishContact() {
  while (Serial.available() <= 0) {
       Serial.println("hello");
       delay(300);
   }
 }

//or if you want to know how much
//is in the buffer
//(buffer holds up to 128 bytes)
byte bytesWaiting = Serial.available;
```

Serial.read()

```
//print what you receive

byte incomingByte;

//using while instead of if for this will
//stick the program here until its done clearing
//the buffer. Can be a better idea to use if's
//and for loops depending what you're up to..

while (Serial.available() > 0) {
   // read the incoming byte:
   incomingByte = Serial.read();
   // say what you got:
   Serial.print("I received: ");
   Serial.println(incomingByte, DEC);
}
```

Serial.flush()

```
//will take the first byte from the buffer...
if (Serial.available() > 0) {
   // read the incoming byte:
   incomingByte = Serial.read();
   // say what you got:
   Serial.print("I received: ");
   Serial.println(incomingByte, DEC);
}
//... and then discards the rest of the
//buffer so it'll be a fresh batch
//the next time you hit this code
Serial.flush;
```

EEPROM

Only have 100,000 rewrites to this. Don't write to it every loop. EVER.

EEPROM Setup

```
//put at top of file
#include <EEPROM.h>
```

no need to attach instance.

EEPROM.write(int *location*, byte *value*);

I'll say it again: Only have 100,000 rewrites to this. Don't write to it every loop. EVER.

location = number between 0 and 512 on ATmega168
location = number between 0 and 1023 on ATmega328

value = full range of byte values, 0-255

```
void eepromClear() {
   for (int i = 0; i < 512; i++) {
     EEPROM.write(i, 0);
   }
}
```

EEPROM.read(int *location*);

location = number between 0 and 512 on ATmega168
location = number between 0 and 1023 on ATmega328

```
void eepromPrintValueAtLocation(int address) {
  byte value = EEPROM.read(address);
  Serial.print(address);
  Serial.print("\t");
  Serial.print(value, DEC);
  Serial.println();
}
```

Servo

Library disables analogWrite() (PWM) functionality on pins 9 and 10 on boards other than the mega. Up to 12 motors on those boards up to 48 on Mega, over 12 will again screw with PWM on Mega pins 11 & 12

Servo Setup

```
//put at top of file
#include <Servo.h>

//declare values
Servo myservo;
int servoPin = 11;

//attach in setup
myservo.attach(servoPin);
```

Servo.write(int *location*);

location = number between 0 and 179, represents degree

```
void servoGoTo(Servo aServo, int location) {
     aServo.write(location);
}
```

also:

Servo.read() : returns last thing you told it to do
Servo. writeMicroseconds() : lets you access the pulsing
Servo.detach() : lets you use PWM again
Servo.attached() : is it currently attached? Returns bool