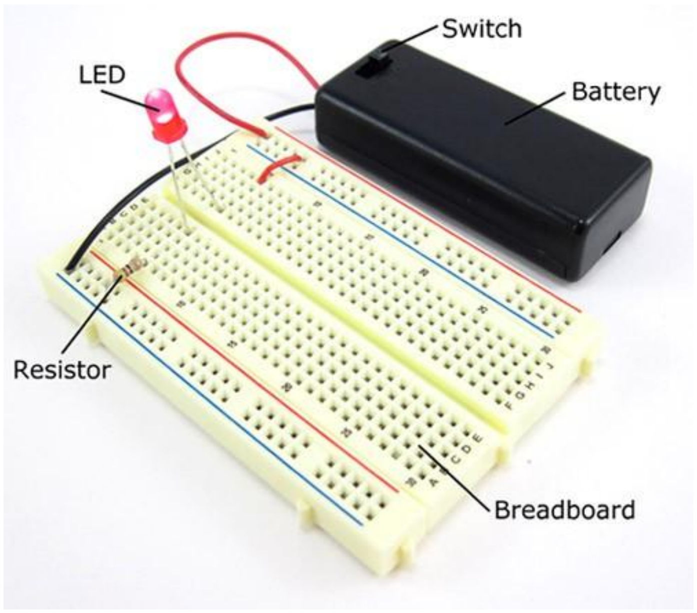
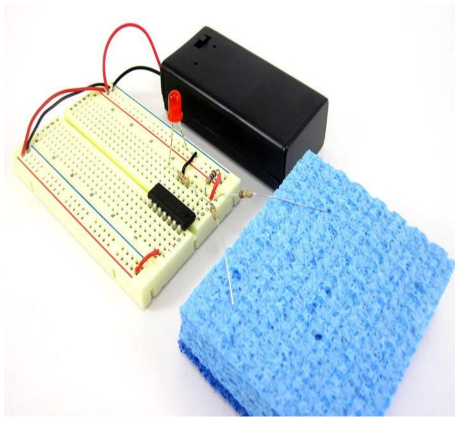
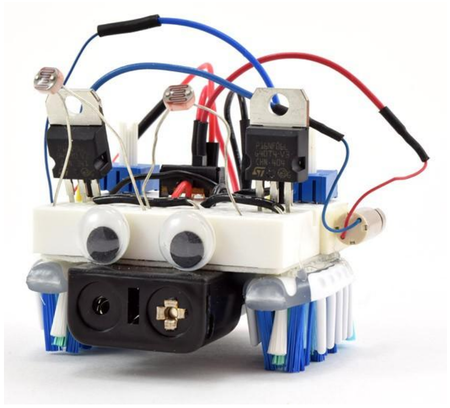
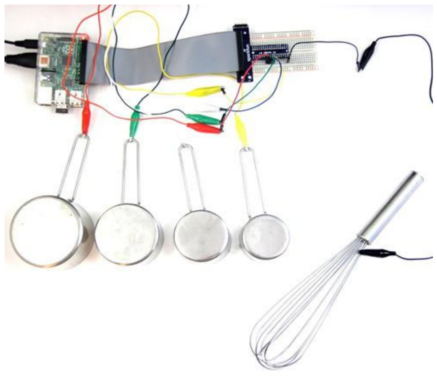
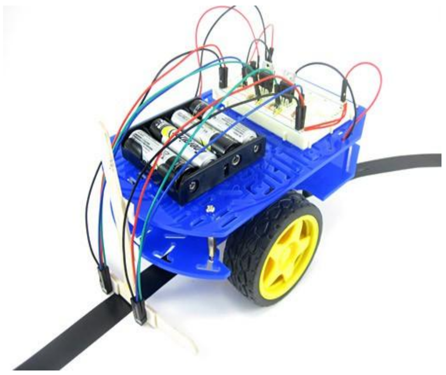
What is a breadboard?

A breadboard is a rectangular plastic board with a bunch of tiny holes in it. These holes let you easily insert electronic components to **prototype** (meaning to build and test an early version of) an electronic circuit, like this one with a battery, switch, resistor, and an LED (light-emitting diode).

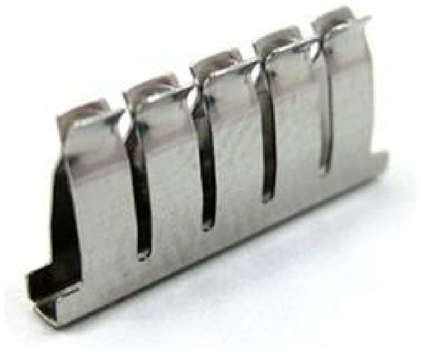


The connections are not permanent, so it is easy to *remove* a component if you make a mistake, or just start over and do a new project. This makes breadboards great for beginners who are new to electronics.

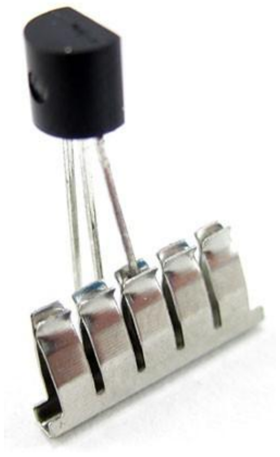





What is inside a breadboard?

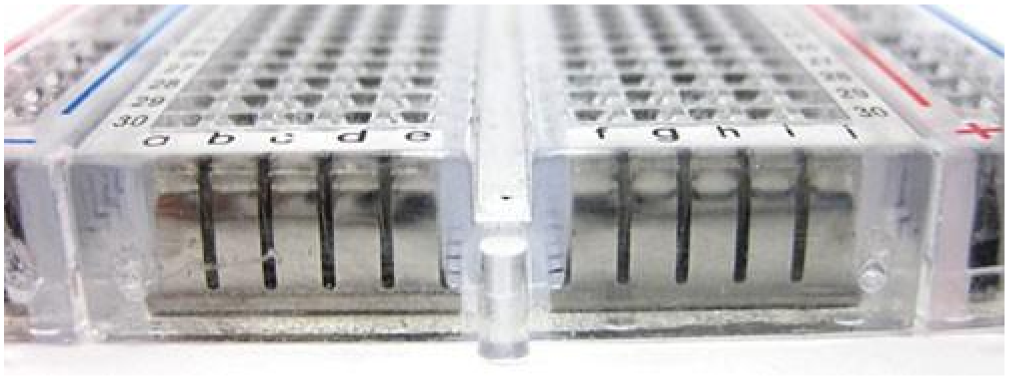
The leads can fit into the breadboard because the *inside* of a breadboard is made up of rows of tiny metal clips. This is what the clips look like when they are removed from a breadboard.



When you press a component's lead into a breadboard hole, one of these clips grabs onto it.



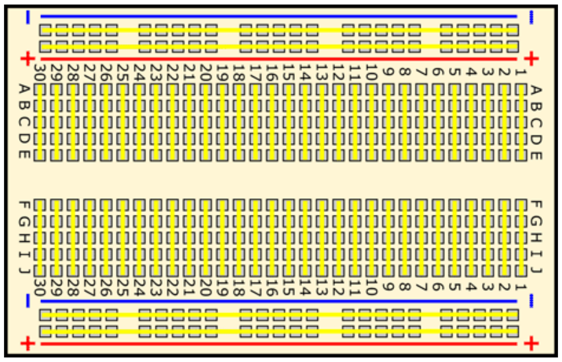
Some breadboards are actually made of transparent plastic, so you can see the clips inside.



Most breadboards have a backing layer that prevents the metal clips from falling out. The backing is typically a layer of sticky, double-sided tape covered by a protective layer of paper. If you want to permanently "stick" the breadboard to something (for example, a robot), you just need to peel off the paper layer to expose the sticky tape underneath. In this picture, the breadboard on the right has had its backing removed completely (so you can see all the metal clips). The breadboard on the left still has its sticky backing, with one corner of the paper layer peeled up.

How are the holes connected?

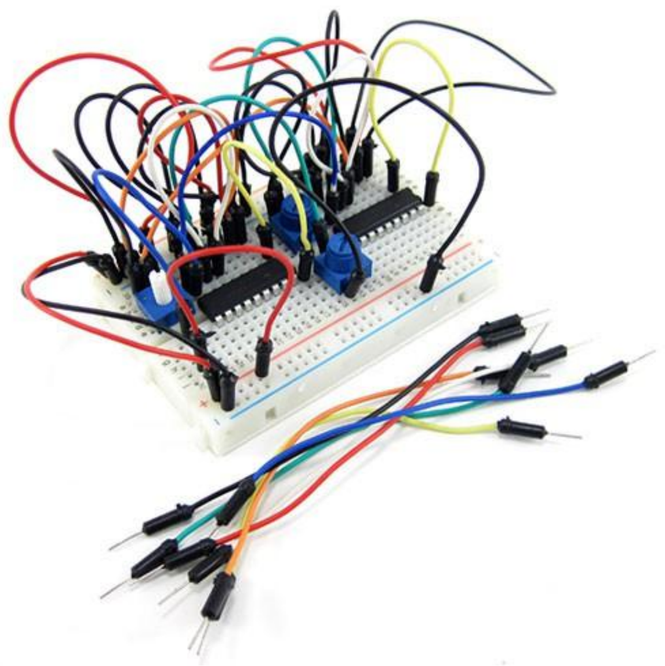
Remember that the inside of the breadboard is made up of sets of five metal clips. This means that each set of five holes forming a half-row (columns A–E or columns F–J) is electrically connected. For example, that means hole A1 is electrically connected to holes B1, C1, D1, and E1. It is *not* connected to hole A2, because that hole is in a different row, with a separate set of metal clips. It is also *not* connected to holes F1, G1, H1, I1, or J1, because they are on the other "half" of the breadboard—the clips are not connected across the gap in the middle of the breadboard. Unlike all the main breadboard rows, which are connected in sets of five holes, the buses typically run the entire length of the breadboard (but there are some exceptions). This image shows which holes are electrically connected in a typical half-sized breadboard, highlighted in yellow lines.



What are jumper wires and what kind should I use?

Jumper wires are wires that are used to make connections on a breadboard. They have stiff ends that are easy to push into the breadboard holes. There are several different options available when purchasing jumper wires.

Flexible jumper wires are made of a flexible wire with a rigid pin attached to both ends. These wires usually come in packs of varying colors. This makes it easy to color-code your circuit . While these wires are easy to use for beginner circuits, they can get very messy for more complicated circuits; because they are so long, you will wind up with a tangled nest of wires that are hard to trace (sometimes called a "rat's nest" or "spaghetti").



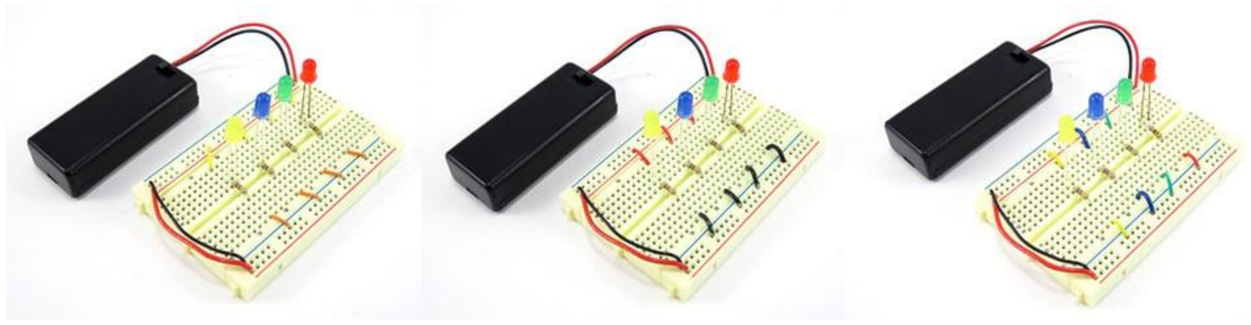
Should I color-code my circuit?

Whether or not you color-code your circuit depends largely on what type of jumper wire you purchase (see the question about jumper wires). Color-coding is a matter of convenience in that it can help you stay more organized, but using different color wires will not change how your circuit works. **Important**: This statement only applies to **jumper wires**. Some circuit components, like battery packs and certain sensors, come with colored wires already attached to them. Keeping track of these colors *does* matter (for example, do not get the red and black leads on a battery pack mixed up). All jumper wires, however, are just metal on the inside with colored plastic insulation on the outside. The color of the plastic does not affect how electricity flows through the wire.

In electronics, it is generally standard to use red wire for positive (+) connections and black wire for negative (-) connections. What other colors you use is largely a matter of choice and will depend on the specific circuit you are building. For example, there are a few different ways you could wire this circuit with red, green, blue, and yellow LEDs, but they will all work exactly the same:

- If you purchased a pre-cut jumper wire kit, use whatever wire colors are available at the appropriate lengths (left image).

- Use red and black wires for the positive and negative sides of each LED, respectively (center image).
- Only use red and black wires for the bus connections, and use red, green, blue, and yellow wire for the respective LEDs (right image).

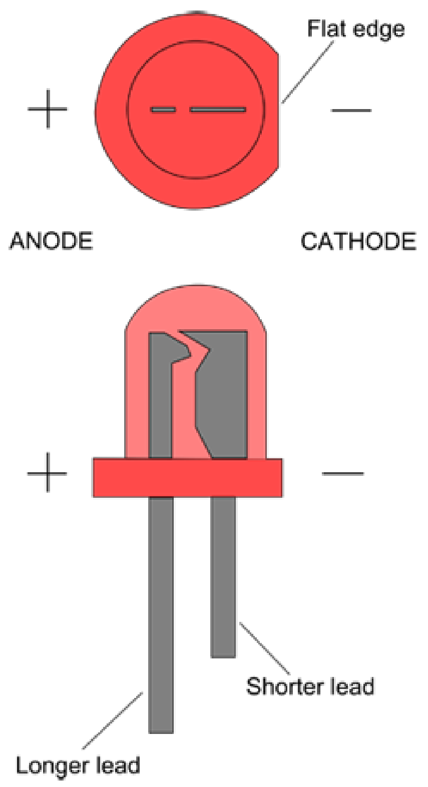


Remember the important part: *the color of the wires does not affect how the circuit works!* The three circuits in this image will all work exactly the same (the LEDs will light up when the battery pack is turned on) even though they have different color wires. If a breadboard diagram shows a blue wire and you use an orange wire instead, nothing will be wrong with your circuit.

Some basic electronic components

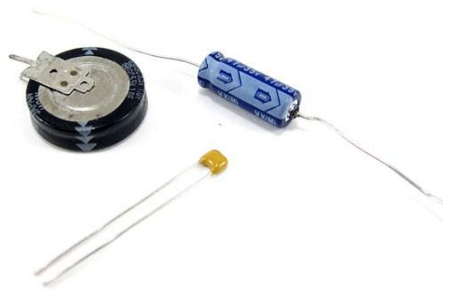
Batteries have a positive terminal and a negative terminal. There are many different types of batteries, but the positive terminal is almost always marked with a "+" symbol. Typically, battery holders will have "+" and "-" symbols printed inside them; make sure the "+" symbols on your batteries line up with the "+" symbols in the battery holder.

LEDs have a positive side (called the **anode**) and a negative side (called the **cathode**). The metal lead for the anode is longer than the lead for the cathode. The cathode side also usually has a flat edge on the plastic part of the LED.

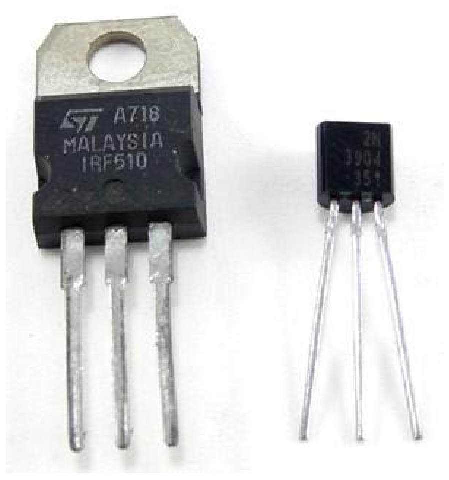


Diodes are like one-way valves that only let electricity flow in one direction. They are usually small cylinders marked with a band or stripe on one end (this is the direction electricity can flow toward).

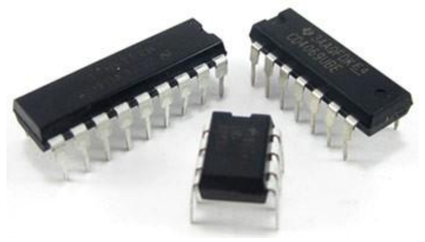
Capacitors are components that can store electrical charge. Common "ceramic disc" capacitors (small orange/tan circles) are not polarized, but several other types of capacitors are, and will typically have arrow or minus signs pointing toward the negative lead.



Transistors are like electronically controlled switches that can be used to turn things like motors and lights on and off. Transistors generally have three pins. Putting a transistor in a breadboard backwards will reverse the order of the pins and prevent it from working. Transistors come in several different "packages," usually a black plastic body with small writing on one side.

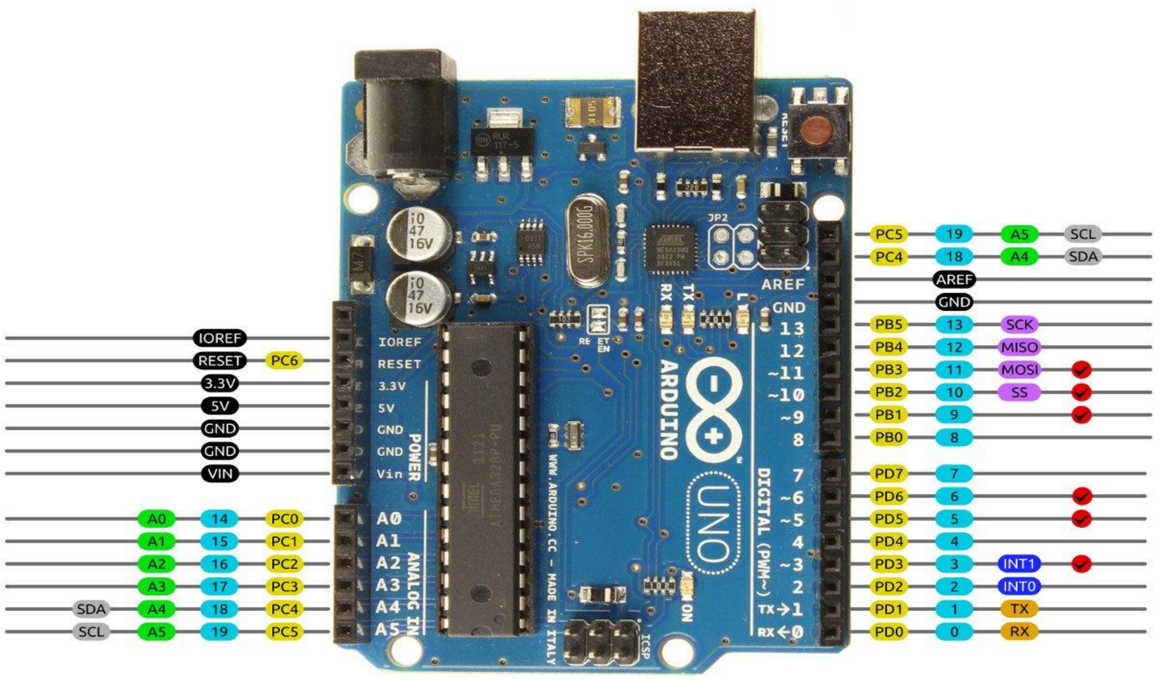


Integrated circuits, or **ICs** for short (sometimes also called "chips") are black rectangular pieces with two rows of pins. They typically have a notch or hole at one end that tells you which way is "up," so you do not put the IC in the breadboard upside-down. See the advanced section on integrated circuits to learn more.



Introduction to Arduino

- Arduino is the development board of AVR microcontrollers.
- The first Arduino board was introduced in 2005 in italy to help students
  who had no previous experience in electronics or mcu programming
  to create working prototypes connecting the physical world to the digital world
- It is an 8-bit device, which means that its data-bus architecture and internal registers are designed to handle 8 parallel data signals.
- **It has nonvolatile Flash memory of 32KB**
  volatile RAM of 2KB and nonvolatile
  EEPROM of 1KB.
- **It has 23 GPIO pins out of which 6 pins can be used for analog and 6 pins can be used for PWM signals.**



Arduino Programming

Language Reference

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

Functions

For controlling the Arduino board and performing computations.

Digital I/O

digitalRead();

[Digital I/O]

Description

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

Syntax

digitalRead(pin)

Parameters

pin: the Arduino pin number you want to read

Returns

HIGH or LOW

Example Code

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

}
```

digitalWrite();

[Digital I/O]

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.

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 Arduino pin number.

value: HIGH or LOW.

Returns

Nothing

Example Code

The code makes the digital pin 13 an OUTPUT and toggles it by alternating between HIGH and LOW at one second pace.

```
void setup() {

  pinMode(13, OUTPUT);    // sets the digital pin 13 as output

}
```

```
void loop() {

  digitalWrite(13, HIGH); // sets the digital pin 13 on

  delay(1000);            // waits for a second

  digitalWrite(13, LOW);  // sets the digital pin 13 off

  delay(1000);            // waits for a second

}
```

pinMode();

[Digital I/O]

Description

Configures the specified pin to behave either as an input or an output. See the Digital Pins page 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 Arduino pin number to set the mode of.

`mode`: `INPUT`, `OUTPUT`, or `INPUT_PULLUP`. See the Digital Pins page for a more complete description of the functionality.

Returns

Nothing

Example Code

The code makes the digital pin 13 OUTPUT and Toggles it HIGH and LOW

```
void setup() {
  pinMode(13, OUTPUT);    // sets the digital pin 13 as output
}


void loop() {
  digitalWrite(13, HIGH); // sets the digital pin 13 on
  delay(1000);            // waits for a second
  digitalWrite(13, LOW);  // sets the digital pin 13 off
  delay(1000);            // waits for a second
}
```

Analog I/O

analogRead();

[Analog I/O]

Description

Reads the value from the specified analog pin. Arduino boards contain a multichannel, 10-bit analog to digital converter. This means that it will map input voltages between 0 and the operating voltage(5V or 3.3V) into integer values between 0 and 1023. On an Arduino UNO, for example, this yields a resolution between readings of: 5 volts / 1024 units or, 0.0049 volts (4.9 mV) per unit. See the table below for the usable pins, operating voltage and maximum resolution for some Arduino boards.

Syntax

`analogRead(pin)`

Parameters

`pin`: the name of the analog input pin to read from (A0 to A5 on most boards, A0 to A6 on MKR boards, A0 to A7 on the Mini and Nano, A0 to A15 on the Mega).

Returns

The analog reading on the pin. Although it is limited to the resolution of the analog to digital converter (0-1023 for 10 bits or 0-4095 for 12 bits). Data type: `int`.

Example Code

The code reads the voltage on analogPin and displays it.

```
int analogPin = A3; // 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();

[Analog I/O]

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 rectangular wave of the specified duty cycle until the next call to `analogWrite()` (or a call to `digitalRead()` or `digitalWrite()`) on the same pin.

Syntax

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

Parameters

`pin`: the Arduino pin to write to. Allowed data types: `int`.

`value`: the duty cycle: between 0 (always off) and 255 (always on). Allowed data types: `int`.

Returns

Nothing

Example Code

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
}
```

Time

delay()

[Time]

Description

Pauses the program for the amount of time (in milliseconds) specified as parameter. (There are 1000 milliseconds in a second.)

Syntax

`delay(ms)`

Parameters

`ms`: the number of milliseconds to pause. Allowed data types: `unsigned long`.

Returns

Nothing

Example Code

The code pauses the program for one second before toggling the output pin.

```
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

}
```

delayMicroseconds();

[Time]

Description

Pauses the program for the amount of time (in microseconds) specified by the 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; larger values can produce an extremely short delay. 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. Allowed data types: `unsigned int`.

Returns

Nothing

Example Code

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

```
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

}
```