



## A beginner's guide to Arduino

by [tttapa](#) on December 26, 2015

### Table of Contents

A beginner's guide to Arduino	1
Intro: A beginner's guide to Arduino	3
Step 1: Before you begin...	3
How not to fry your Arduino	3
Step 2: Software	4
Arduino IDE	4
Arduino IDE + Teensyduino	4
Step 3: Hardware & Electronics	5
Basic physics	5
Resistors	5
Capacitors	5
Transistors	5
MOSFETs	6
Diodes	6
LEDs	6
Relays	6
Other parts	6
Power	7
Storage	7
Tools	7
Step 4: Blink: Digital outputs	11
Blink: first Arduino code	11
Step 5: Uploading a program to the Arduino	13
Step 6: Driving more LEDs	13
Calculating the resistor values for our LEDs	13
Connecting the LEDs to the Arduino	14
More LEDs	14
Optimizing the code	14
Summary:	15
Extra: Calculating the power dissipation in the resistor	15
Step 7: Even more blinking LEDs	16
Programming the LED sequence	16
Summary:	17

Extra: 2-dimensional arrays (Matrices)	17
Step 8: Input from a button	19
Pull-up and pull-down	19
Internal pull-up resistors	19
Summary	19
Extra: Direct port manipulation	19
Step 9: Communication with the computer	22
Sending inputs to the computer	23
Sending commands to the Arduino	23
Summary	24
Extra: Switch	24
Step 10: Analog inputs and outputs.	25
Analog Inputs	25
Potentiometers	25
Measuring voltages	25
Analog outputs	26
Step 11: Driving more LEDs or other loads	28
Measuring voltage and current	28
Calculating the base resistor for the transistor	28
Low-current MOSFETs	28
High-current MOSFETs	29
Relays	29
Summary	29
Extra: PNP transistors	29
Step 12: Driving motors	34
DC motors	34
Stepper motors	34
Step 13: Things not (yet) covered in this Instructable	37
The community	37
Step 14: Final thoughts	37
Related Instructables	37
Advertisements	38
Comments	38

## Intro: A beginner's guide to Arduino

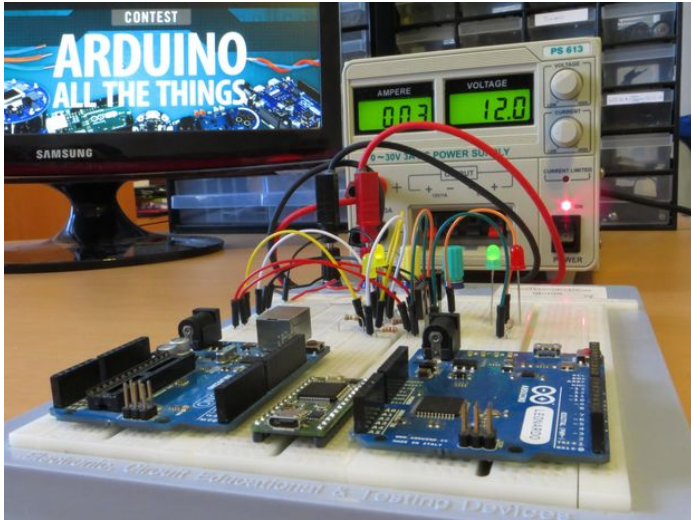
After some years of experimenting with Arduino, I decided that the time has come to share the knowledge I've acquired. So I here it goes, a guide to Arduino, with the bare basics for beginners and some more advanced explanations for people who are somewhat more familiar with electronics.

Every step will consist of a detailed explanation, then a summary, followed by a more advanced approach.

If you're a complete beginner, I recommend reading the explanation first, and then the summary. There will (almost certainly) be some parts of the explanation you don't understand. Don't worry, it is absolutely normal, it will become clear after reading some other examples in the next steps, just don't get discouraged!

I'll provide some links to the Arduino reference page, Wikipedia, and some other interesting sources. If you want to know more about a certain subject, these are great places to start. Again, if you don't understand a word of those links, don't worry, it is absolutely not necessary for this Instructable, and especially for beginners, they can be very confusing or even demotivating. If that's the case, it might be better to skip them for now. But don't give up!

Although a tutorial like this might be very helpful, you'll learn mostly by experimenting yourself. Use this Instructable as a starting point, as a reference, but make your own projects, modify the given examples, try new things, search the internet, the possibilities are pretty much endless!



## Step 1: Before you begin... How not to fry your Arduino

Before you begin plugging things into your new Arduino, it may be good to know what can damage the board.

1. Drawing more than 40mA from an output pin.  
An Arduino can only supply 40mA per output pin, so you cannot drive a motor or a speaker directly, for example, and you cannot connect an LED directly (without a resistor) in the course of this Instructable, I'll explain what you should do instead.  
Shorting an output in to the +5v or the ground, will also kill your board: If an output pin is at 5v for example, and you connect it to the ground, it draws an enormous amount of current, and kills your Arduino almost instantly.  
The pins go through the circuit board, so make sure you don't place the Arduino on a conductive (metal) surface, because it will short out the pins.
2. Drawing more than 200mA from all output pins together.  
The ATmega chip on your Arduino can only supply 200mA in total, so driving more than 10 LEDs @ 20mA each, for example, will eventually damage your board.
3. Supplying more than 5v (3.3v) to an input pin.  
Supplying more than the operating voltage of the Arduino on any pin is very dangerous. Some Arduinos that run at 3.3v have 5v tolerant pins, but that's about it.
4. Supplying more than 5v to the 5v pin.  
The 5v of the Arduino board goes directly to the ATmega chip, that is rated for an absolute maximum of 6v.
5. Supplying more than 12v to the Vin pin.  
There's an onboard 5v voltage regulator on the board, that will overheat if you feed it more than 12v.
6. Drawing more than 500mA from the 5v pin (when running off an external power supply).  
The onboard 5v voltage regulator can only supply 500mA of current. The 5vUSB has a polyfuse to limit the current to 500mA.
7. Drawing more than 50mA from the 3.3v pin.  
The onboard 3.3v voltage regulator can only supply 50mA of current.
8. Reversing the polarity of the power supply.  
If you swap the 5v or Vin pin with the GND pin, you'll kill the board almost instantly.  
The barrel jack has a diode to protect against reverse polarity.
9. Connecting a load to the Vin pin while using USB power.  
If you connect a load to the Vin pin while the 5v to the Arduino comes from the USB connection, current will flow backwards through the voltage regulator, damaging it.
10. Static electricity  
Although most chips have clamping diodes as protection against ESDs (electrostatic discharges), it may be wise to use an anti-static wrist strap, or to remove the carpet under your desk.

## Step 2: Software

### Arduino IDE

For programming our Arduino, we'll need the Arduino IDE (integrated development environment). It can be downloaded from the [site](#).

Windows:

1. Go to the site, go to Download, and select the Windows installer.
2. Consider donating some money, if you want to support the project, and click download.
3. Click the downloaded file to run it.
4. Grant administrator permission.
5. Agree to the License Agreement.
6. Select the appropriate boxes, you'll need the Arduino Software and the USB driver, the other three are optional.
7. Accept the default installation folder, or pick another one. Then click install.
8. When you're prompted whether or not you want to install the Arduino USB Driver (device software), click install.
9. Wait for the installer to complete, and launch the Arduino IDE.

Ubuntu:

1. Go to the site, go to Download, and select the right Linux version.
2. Consider donating some money, if you want to support the project, and click download.
3. Open a terminal window (CTRL+ALT+T), and run these commands, changing the filename appropriately
4. `cd Downloads`
5. `tar xf arduino-1.6.7-linux64.tar.xz`
6. `sudo mv arduino-1.6.7/ /opt/arduino-1.6.7/`
7. `/opt/arduino-1.6.7/install.sh`
8. `sudo usermod -a -G dialout $USER`

This goes to the Downloads folder, unzips the downloaded file, moves it to the /opt/ folder, and runs the install script. This install script will create a desktop file, and a MIME type to associate .ino files with the Arduino IDE. Finally, you have to add (-a = append) your user to the 'dialout' group (-G), in order to have access to the serial ports. (\$USER is a system variable for the current user) If you open the Arduino IDE now, you'll see that the Tools > Port option is grayed out. When your user is added to the dialout group, log out, and sign back in. Your Arduino's serial port should now be available in the IDE.

### Arduino IDE + Teensyduino

If you have a Teensy board, you'll need the Teensyduino add-on for the Arduino IDE. The installation is very simple, and a very good install guide can be found on the [Teensy site](#).

Teensyduino currently doesn't support the latest version (1.6.7 @27-12-2015) of the Arduino IDE yet, so you'll have to download the previous version. (1.6.6)

If you're running Linux, you may come across this error while installing Arduino:

```
bash: /opt/arduino-1.6.6/install.sh: Permission denied
```

If this is the case, try running

```
sudo chmod +x /opt/arduino-1.6.6/install.sh
```

This adds (+) the permission to be executed (x). Then try running /opt/arduino-1.6.6/install.sh again.

The **examples** I use can be found [here](#).



### Step 3: Hardware & Electronics

Before we begin, I'll explain some of the basic electronic components. **If you only just started with electronics, this is for you!**

Sometimes I'll use some physics to explain how a certain component works, this is just a side note, it doesn't matter if you don't understand this yet.

I also provided some links to videos on YouTube that helped me understand the basic principles of the different components.

#### Basic physics

Electricity is the flow of electric charge carriers: electrons (in most cases).

Electrons are the negatively charged particles that whirl around the positively charged nucleus (core, plural: nuclei) of an atom.

Electrons can move easily through metals, like copper, silver, gold ... We call these materials conductors. These materials have freely moving electrons.

Materials like plastic, wood, glass, air ... don't conduct electricity very well. They are called insulators. They don't have moving electrons or other charge carriers.

A piece of material that has more negative charges (electrons) than positive ones (nuclei with positive protons), is negatively charged. A piece of material that has less negative charges than positive ones, is positively charged. (Note that only the electrons can move, the positive nuclei are stuck in a grid.)

Just like magnets, opposite charges attract each other: when you have one piece of material that has more electrons, and one piece that has less electrons, the electrons in the negative piece will be attracted to the positive piece. If there's a conductor in between these pieces, these electrons will 'flow' to the positive part: This is electric current.

Current expresses the amount of charges that flow through a conductor per unit of time. Its unit is Amps (Ampère), and is defined as C/s, where C is Coulomb (charge) and s is seconds (time). Its symbol is I.

A battery has a negative side that has more electrons, and a positive side that has fewer electrons. Like I said earlier, the electrons will try to reach the positive side, but they cannot go through the internal circuit of the battery itself. This gives the electrons potential energy. This is the energy that is released as light and heat in a bulb, as motion (kinetic energy) in a motor ... The difference in potential energy of a charge at the positive and a charge at the negative side, is called the voltage. The unit is Volts, and is defined as J/C, where J is Joule (SI-unit of energy) and C is Coulomb (SI-unit of charge). This expresses how much energy a certain charge (read: certain amount of electrons) releases. The symbol for Volts is V or U (from the German word 'Unterschied', difference, and refers to the potential difference).

Power is the amount of energy that is released per unit of time. The SI unit is Watts, and is defined as J/s where J is Joules, and s is seconds. If you multiply current by voltage (C/s \* J/C) the C cancels out, so you get J/s. This means that voltage multiplied by current gives you the wattage.

In most schematics, the conventional current flow is used: arrows are drawn from the positive side to the negative side. In practice, however, only electrons can move, so the actual direction of the current flow is from the negative side to the positive side.

#### Resistors

Resistors are components with - as the name implies - an electrical resistance, in other words, they limit the flow of electrons, so they are often used to limit the current.

The SI unit of resistance is Ohms, often written as the Greek letter omega ( $\Omega$ ). They are often used with the unit prefixes kilo (k) and mega (M). E.g.  $1.2M\Omega = 1M2\Omega = 1,200k\Omega = 1,200,000\Omega = 1,200,000E\Omega = 1,200,000R\Omega$ . (note that writing a digit after the unit prefix is the same as writing it after the decimal point. Also, in some schematics, E or R are used instead of  $\Omega$ ).

The value of a resistor is indicated by 4 (or 5) colored bands, using the resistor color code:

The first 2 (or 3) bands are the 2 (or 3) first digits of the value, and the 3rd (or 4th) band is the power of ten that comes after those 2 (or 3) digits. This is also called the multiplier, and is just the number of zeros you have to add. The last band is the tolerance, and is mostly silver or gold. E.g. red red red gold =  $22 \times 100\Omega = 2,200\Omega = 22 \times 10^2\Omega = 2k\Omega = 2.2k\Omega$ , with a tolerance of 5%; green blue black brown red =  $560 \times 10\Omega = 5,600\Omega = 5k6\Omega = 5.6k\Omega$ , with a tolerance of 2%.

The relationship between resistance, voltage and current can be calculated using **Ohm's Law**.

$$I = V/R$$

where I is the current in Amps, V the voltage in Volts, and R the resistance in Ohms. This is a very, if not the most important formula in electronics, so try to remember it!

#### Capacitors

A capacitor is an electrical component that can store electrical charge (in the form of electrons).

Although they are fundamentally different, in some ways, it behaves like a small rechargeable battery.

When a voltage is applied to a capacitor, the potential difference (a difference in number of electrons) the side with more electrons has a negative charge, compared to the other side. These electrons can flow out of the capacitor again, when the voltage is no longer applied, just like a battery.

Capacitors are used in filters, for example to filter out the 50/60Hz noise from your power supply, or to filter high frequencies out of your music when you turn on the low-pass filter, or turn the bass and treble knobs on your amplifier. In these cases, the capacitor charges and discharges really quickly. Another use for the capacitor, is filtering out DC voltage.

The SI unit of capacitance is Farad, or F. This is a very large unit, and most often, you'll see prefixes like pico (p), nano (n) or micro ( $\mu$ ).

On some smaller capacitors, the capacitance is written using a three-digit number. The first two digits are the first two digits of the value, and the third digit is the power of ten to multiply it with. The unit of the value you get is picofarad.

E.g.  $104 = 10 \times 10^4 = 100,000 \text{ pF} = 100 \text{ nF} = 0.1 \mu\text{F} (= 0.0000001 \text{ F})$

Larger capacitors, the electrolytic type, (mostly the cylindrical ones) have a polarity, marked by a grey line. If you connect them the wrong way around, they can explode, be careful!

#### Transistors

<http://www.instructables.com/id/A-Beginners-Guide-to-Arduino/>

A transistor is a semiconductor device, that is used to switch or amplify a signal. You can think of it as a switch, that can be operated by using a very weak signal, a current controlled switch.

A transistor has three terminals: they are called the base (B), the emitter (E) and the collector (C).

The emitter 'emits' electrons, and they are 'collected' by the collector. The base is used to control this flow of electrons.

If a small current flows from the base to the emitter, a much larger current will flow from the collector to the emitter. How much larger this C-E current is, depends on a constant, specific to the type of transistor. This constant is called the DC current gain, and has the symbol of the Greek letter beta ( $\beta$ ) or  $H_{fe}$ .

E.g. if you have a transistor with  $\beta = 100$ , and your B-E current = 10mA, your C-E current will be 1A.

This principle is used in amplifiers.

Obviously, the transistor cannot keep on amplifying forever: at a certain point, the transistor will just act like a switch: the transistor is now in saturation mode.

There are two types of transistors: NPN and PNP. This has to do with the semiconductors inside.

The difference is the direction in which the current flows, more on this in the examples in the following steps.

## MOSFETs

Another type of transistor is the MOSFET, acronym for Metal Oxide Semiconductor Field Effect Transistor.

The MOS just stands for the materials it is made of, and FET signifies that the amount of current that is let through is controlled by a field, an electric field, more specifically. Physics tells us, that the higher the voltage, the stronger the electric field, so we can control the current using a voltage, whereas the normal (Bipolar Junction Transistor or BJT) uses current to control the current.

A MOSFET also has three pins: a gate (G), a drain (D) and a source (S).

The source is where the electrons come from, and they flow to the drain. This flow is controlled by the voltage at the gate (and its accompanying electric field). By analogy with the transistor, the gate can be compared to the base, the source to the emitter, and the drain to the collector.

An advantage of a MOSFET over a BJT is the higher efficiency: when fully turned on, a MOSFET has a D-S resistance of a few tens of milliohms. This results in much less power (heat) dissipation when driving high-current loads.

Also, no current flows from the gate to the source.

A disadvantage though, is that you need about 10V on the gate for most MOSFETs to be fully on. This is 2-3 times higher than the voltage of an Arduino output pin, for example.

## Diodes

Just like a transistor, a diode is a semiconductor device. One of the interesting properties of a diode, is that they only conduct electricity in one direction.

For example, Arduino boards have a diode in series with their power input jack, to prevent you from reversing the power, and damaging the chip.

Diodes have a forward voltage drop ranging from 0.5V to 0.7V. This means that if you measure the voltage before the diode, it will be about 600mV higher than after the diode.

Of course, a diode has its limits: if the reverse voltage is too high, it will break, causing it to let current pass in the wrong direction. In some diodes, this is done in a controlled way. These diodes are called zener diodes. They will only conduct if the voltage is higher than a certain value, specific to the zener.

This value is constant, so zener diodes are used as a reference in voltage regulators.

## LEDs

An LED, acronym for Light Emitting Diode, is like a normal diode, but they emit the energy (of their forward voltage drop) as light, instead of heat. Their voltage drop is higher than a normal diode: from 1.2V for an infrared LED, up to 3.5V for blue and ultraviolet LEDs.

If the current going through the LED is too high, it will die. To prevent this, a resistor in series is used.

Always do this, otherwise, you'll kill the LED within a second.

## Relays

A relay is a true current-controlled switch. It consists of a coil, next to a piece of metal, that is pulled back by a spring. When current flows through the coil, it generates a magnetic field that attracts the piece of metal, and makes a connection.

The advantage is that you can control very high-current or AC loads, and they add virtually no extra resistance.

The disadvantages are that relays are slow, since they have to move physically, they are more fragile, due to the moving parts, they are extremely slow, compared to a transistor, and they can create sparks.

## Other parts

Of course there are countless other components you can use in your Arduino projects:

Microphones and speakers: Dynamic microphones have a coil and a magnet to convert the vibrations of the air to electrical signals. Similarly, speakers use a coil that moves in a permanent magnetic field to generate those vibrations, when fed with an AC signal. Electret microphones translate air movement to changes in capacity. Piezo disks convert vibration to voltage, and vice versa, so they can be used as both a mic and a small speaker.

Switches: switches are easy input devices for your Arduino, they exist in all shapes and sizes.

Variable resistors or potentiometers: this is just circular resistive trace, and a wiper, connected to a turning shaft, that changes the resistance as it moves along the trace. Small versions without a shaft are called trimpots.

ICs and chips: There's an immensely wide variety of ICs available, like voltage regulators, microprocessors, op-amps, amplifiers, logic gates, memory, timers, and so on.

Sensors: You can find a sensor for virtually anything, light sensors, temperature sensors, distance sensors, alcohol sensors, even GPS modules, cameras ... Other variants are optointerrupters, reed (magnetic) switches ...

Rotary or optical encoders: they convert movement to a series of pulses, like the volume knob in your car, or knob on your microwave oven.

Displays: LCD displays can be used (some with touchscreen), or simple 7-segment LED displays, even small OLED displays are available.

Fans, coils and motors: computer fans, solenoids, DC motors, stepper motors, servos, and so on.

## Power

You can power your Arduino from a USB port, but this solution is limited to 5v and only 500mA, so if you want to use things like motors, or things that require a higher voltage, you'll need a power supply.

A benchtop power supply is the best solution, I think: They have current limiting features, adjustable voltages, and they can deliver a lot of power. Most of them also have some convenient 12v and 5v output, besides their adjustable output. But they tend to be quite expensive...

A solution can be a wall-wart adapter, that plugs right into your Arduino. The on-board voltage regulator of the Arduino will step it down to 5v for the chip itself. The regulator can take any voltage between 6v and 12v, according to the specs.

Another great power solution is a computer power supply: they have lots of power, thermal protection, short circuit protection, and deliver the most common voltages (3v3, 5v, 12v). There are loads of Instructables on how to hack an old computer PSU, for example: <http://www.instructables.com/id/A-Makers-Guide-to-...>

A disadvantage is that the current protection is not sensitive at all, since it is designed for computer components that can draw over 30A or more in total, so your circuit may explode and catch fire, destroying anything that it's connected to, as long as it draws less than the rated current, the PSU will happily keep on supplying power. Also, the PSU uses really high voltages, inside a metal case, so hacking it isn't without any risks...

You could also build your own power supply of course, but it will probably be cheaper to just buy a decent benchtop power supply.

Power sources for mobile applications can be coin cell batteries, if the circuit doesn't draw a lot of power, or standard AA batteries, a 9v battery, rechargeable Ni-MH or Li-ion batteries, a USB powerbank, or even solar panels.

## Storage

I use two drawer cabinets to store all small components, and a dozen of other boxes for motors, PCBs, cables etc. Some have small compartments, to store screws, nuts and bolts.

If your Arduino or some other IC or chip came in a shiny plastic bag, don't throw it away! It is probably an antistatic bag, to protect components that are prone to damage due to ESD (ElectroStatic Discharge), use them to store your chips.

Also, most ICs come in a piece of antistatic foam, keep them for storing your chips, it protects them against ESD, and keeps the legs from bending.

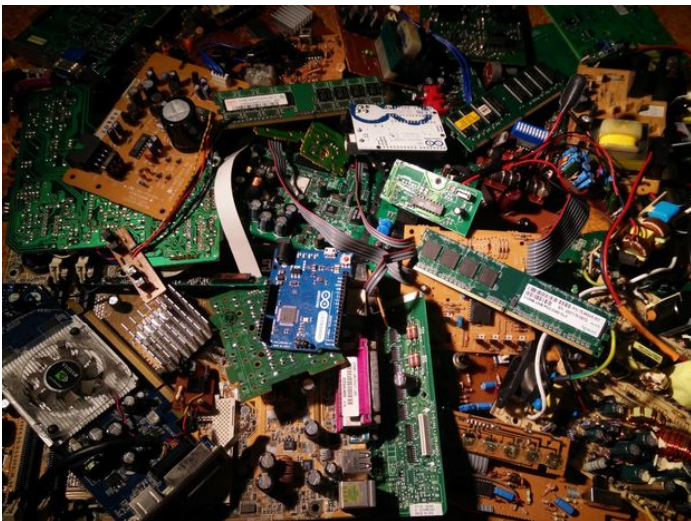
## Tools

The basic tools you'll need are wire cutters and wire strippers, probably some pliers and a set of small screwdrivers. A multimeter comes in handy very often, and if you have two of them, you can measure both voltage and current at the same time, which is a big plus, though not at all necessary.

You'll also need a soldering iron, and some solder, maybe a desoldering pump, to salvage parts from an old PCB.

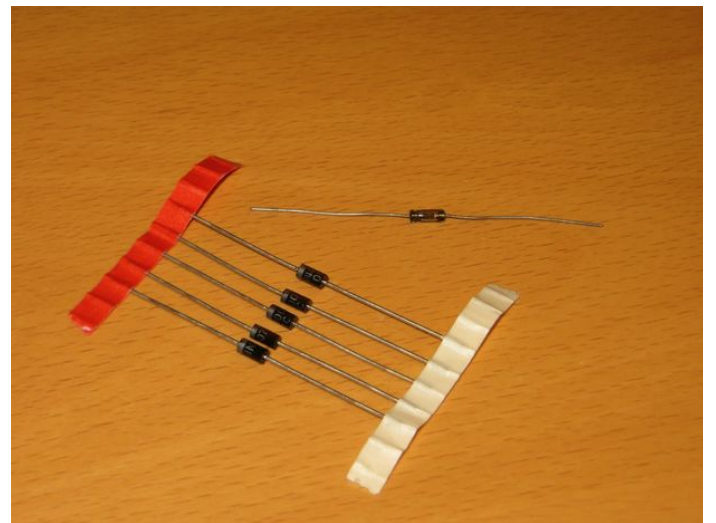
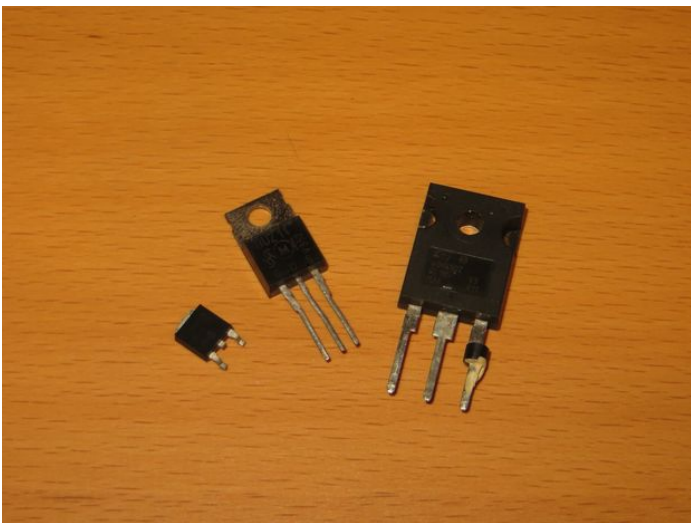
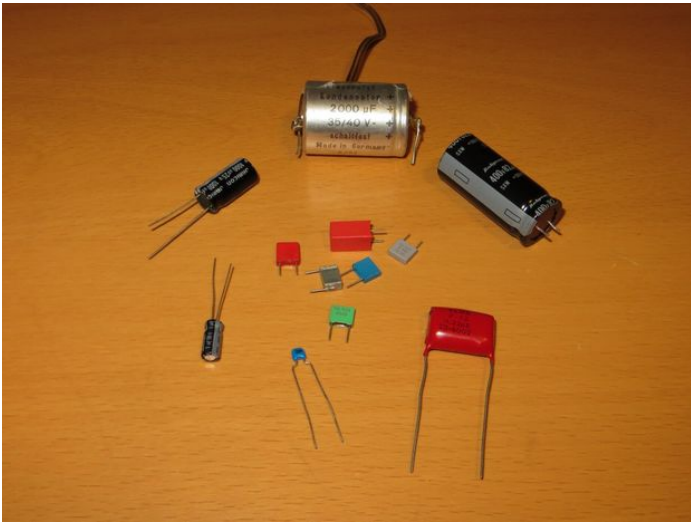
For prototyping, you'll need a **solderless breadboard**, and some jumper wires. You could also use thin copper wire with a solid core. Either way, you'll need some wire, I usually buy red, black and white wire, about 10m each. (Red is used for positive, black for negative or ground, and white for 'other things') You'll be surprised of how fast you use it up.

Some perfboard can come handy for permanent circuits.

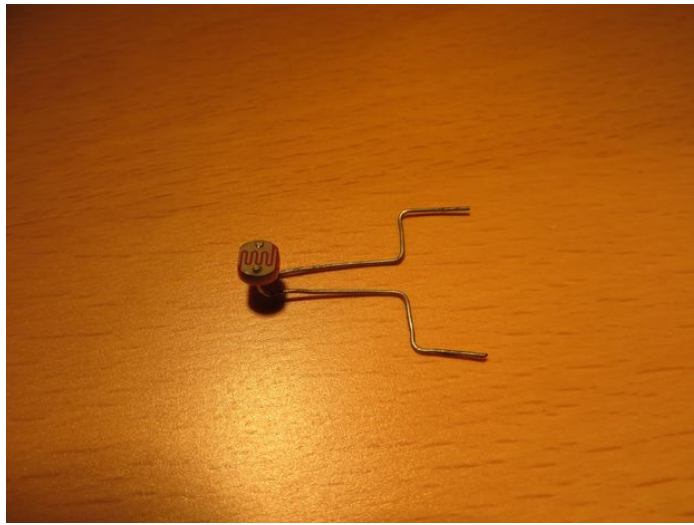
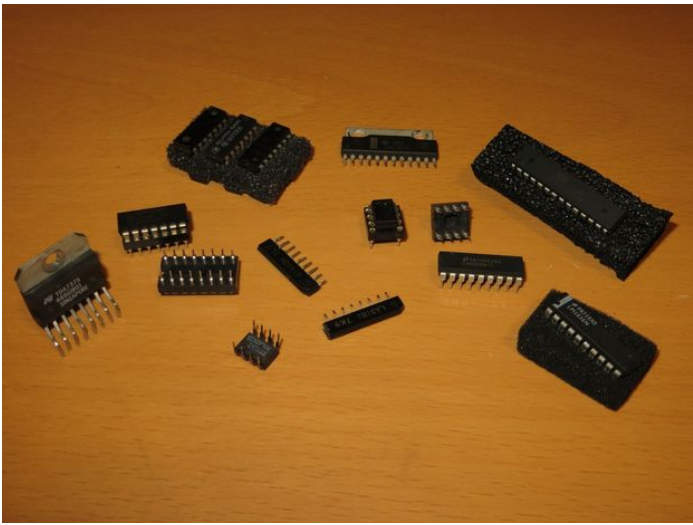
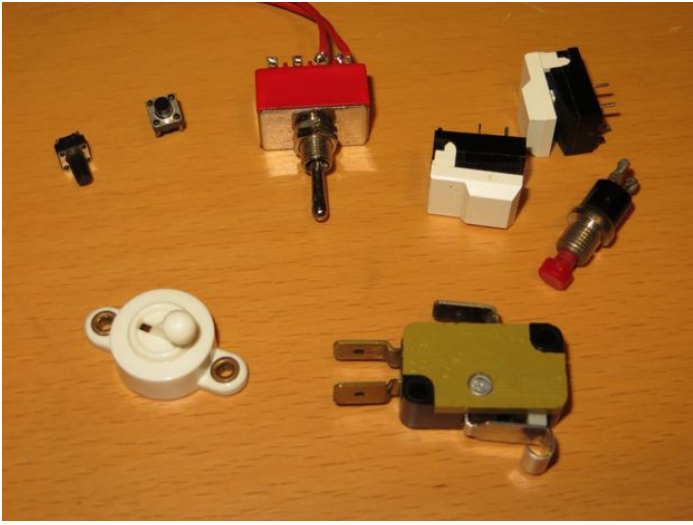


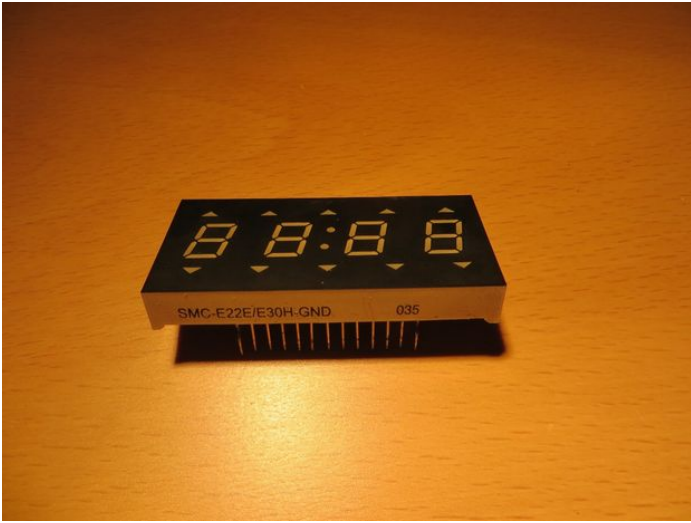
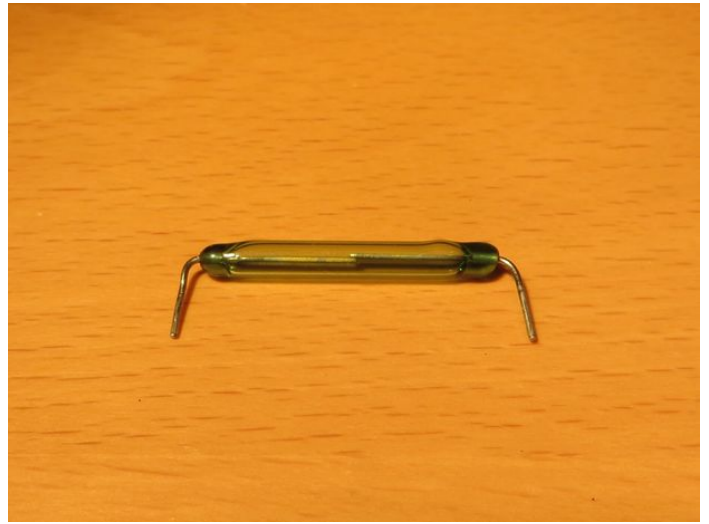
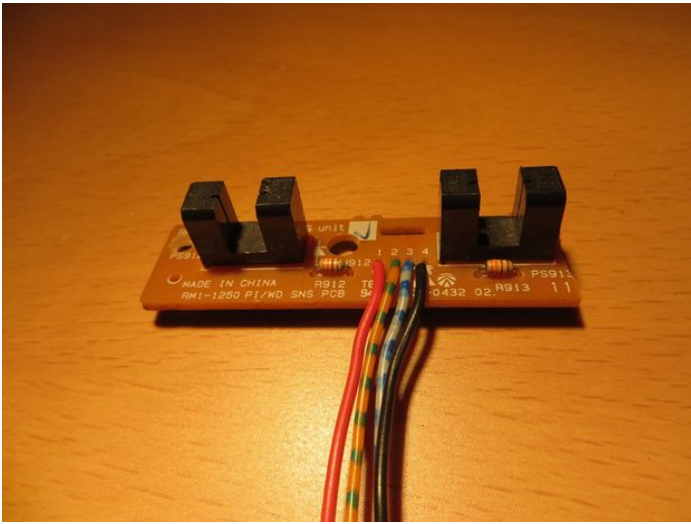


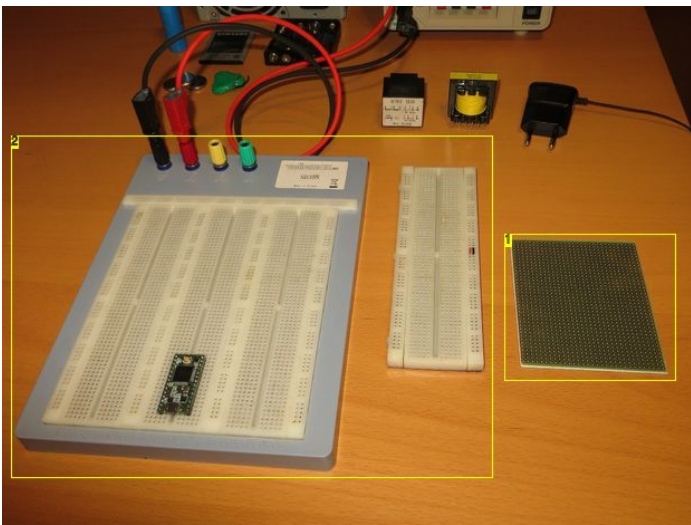
	1 <sup>st</sup> BAND	2 <sup>nd</sup> BAND	3 <sup>rd</sup> BAND	4 <sup>th</sup> BAND
COLOR	(1 <sup>st</sup> digit)	(2 <sup>nd</sup> digit)	(multiplier)	(tolerance)
Black	0	0	1Ω	
Brown	1	1	10Ω	±1%
Red	2	2	100Ω	±2%
Orange	3	3	1kΩ	
Yellow	4	4	10kΩ	
Green	5	5	100kΩ	±0.5%
Blue	6	6	1MΩ	±0.25%
Violet	7	7	10MΩ	±0.1%
Grey	8	8		±0.05%
White	9	9		
Gold			0.1Ω	±5%
Silver			0.01Ω	±10%











#### Image Notes

1. Perfboard
2. Breadboards

## Step 4: Blink: Digital outputs

### Blink: first Arduino code

When you plug in your Arduino for the first time, you'll see a green light (with 'ON' written next to it - this is the power LED) and an orange light that blinks (with 'L' written next to it). This is the default 'Blink' program, it turns the internal LED on for a second, then turns it off for a second, repeating forever.

Let's take a look at the code: Open the Arduino IDE and go to File > Examples > 01.Basics > Blink.

The first thing you'll notice, are the first 14 lines that are lighter than the rest. All text placed between the `*/` signs, is a comment. This is a multi-line comment. On line 17, there are some more comments, they use the `//` operator, everything on that line, that comes after the `//` is a comment. The next line is just normal code again, you don't have to close this single line comment, unlike the multi-line comment.

Comments are used to make your code more readable, so it is strongly recommended that you use them in your own programs. It's not really necessary in a small program, like Blink, but when you are writing code of over a thousand lines, you'll be very thankful if you've added some comments, to help you understand what a particular piece of code does, and how it works.

You can also use the comment operators to (temporarily) disable a piece of code, without having to delete it permanently. This works, because comments do not get uploaded to the Arduino. (They are ignored by the `compiler`, the piece of software that converts the code you write in the Arduino IDE to commands that the Arduino can understand.) This also means that they do not take up any space in the Arduino's (limited) memory, only on your computer's hard disk.

Arduino reference: Comments

The next bit of code we encounter is `void setup()`!

This is the setup routine, it runs only once, every time the Arduino is started up. (void is a data type, it means no information is returned by the setup, more on this later. The two brackets are used in every function, to pass data through; setup doesn't need any data, so the brackets are empty. Don't worry if you don't understand this yet, we'll cover it in detail in the step about functions.)

Everything between the two curly brackets or braces after `void setup()` is the code that executes during the setup. Make sure you always close your brackets, otherwise, you'll get strange errors. The IDE helps you with this by highlighting the other bracket, if you select one.

Arduino reference: void

Arduino reference: setup

Arduino reference: curly braces

The first real command is `pinMode(13, OUTPUT)`!. As you might have guessed, this has to do with the two rows of pins on every Arduino: Those pins can be used either as an input or as an output. The first number is the pin we want to use as an output, 13 in this case, since all Arduino boards have an onboard LED connected to pin 13. OUTPUT, the second argument of the `pinMode` function, is a constant. This is a value that is defined by the software itself, and has been given an easy name. Constants will have a blue color.

(The value of OUTPUT is 1, which is a bit counterintuitive, since its O looks like a 0, and the I of the INPUT constant looks like a 1. `pinMode(13,OUTPUT)` exactly the same as `pinMode(13,1)` )

By default, the Arduino's pins are set as an input, but by using the `pinMode` function, we've now set pin 13 as an output. We haven't told it what value we want to output yet, so it will just be 0. This means that, inside the chip, pin 13 is connected to 0v, this is the ground. If you would connect the 5v pin to pin 13 now, you would create a **short circuit! Be careful!**

Arduino reference: pinMode

Arduino tutorials: Digital Pins

In the Arduino IDE, and other C-like programming languages, every statement is ended with a **semicolon** (`;`), as you can see in this Blink example. When you get cryptic errors when trying to upload, one of the first things to check are the semicolons. Most of the times, you'll get an error like this: Blink:16: error: expected ',' or ';' before... 16 is not the line where there's a missing semicolon, but the line with the next command/statement.

Arduino reference: ; semicolon

The next structure element is `void loop()`!

The code between the curved brackets is executed after the setup is finished, and will repeat forever (at least until you restart the Arduino, or upload another program).

<http://www.instructables.com/id/A-Beginners-Guide-to-Arduino/>

Arduino reference: loop

The next function is '**digitalWrite(13, HIGH)**'

This 'writes' pin 13 high, in other words, it connects it internally to the 5v pin. The LED on the Arduino is connected between the ground and pin 13, so now there's 5v across the LED, and it will light up.

Be careful not to short pin 13 to the ground now, it will create a short circuit!

You can see the same function being used with the LOW constant as well, this will connect pin 13 to the ground (inside the chip). Now there's no voltage difference across the LED, and it will go out.

Instead of HIGH and LOW, you could also use 1 and 0, or true and false.

Arduino reference: digitalWrite

The last function is a very intuitive one: **delay(1000)**

It just waits for a certain amount of time, in milliseconds. In this case, it will wait 1000 ms, or 1 second, before executing the next command.

Arduino reference: delay

When we arrive at line 29, we start all over at line 25, because we're in the loop.

### Summary:

1. In the setup, that only runs once when the program is started, we set pin 13 as an output.
  2. In the loop, we make the output of the led HIGH (5v), wait 1,000ms, make it LOW (0v) and wait for another second. This loop will be repeated forever (at least until you restart the Arduino, or upload another program)
- /\*this is a comment \*/ this is not
  - this is not a comment // this is a comment
  - every statement ends with a semicolon ;
  - void setup(){ } is the function that runs once, when the Arduino starts up
  - void loop(){ } is the function that repeats forever, after the setup has run
  - pinMode(pin,OUTPUT); or pinMode(pin,1); sets the given pin as an output
  - digitalWrite(pin, state); sets a given pin high (5v) or low (0v). State can be HIGH or LOW, 1 or 0, true or false.
  - delay(time); waits for a given amount of time, in milliseconds.



```
Blink | Arduino 1.6.7
File Edit Sketch Tools Help
Blink
1 | /*
2 |  Blink
3 |  Turns on an LED on for one second, then off for one second, repeatedly.
4 |
5 |  Most Arduinos have an on-board LED you can control. On the Uno and
6 |  Leonardo, it is attached to digital pin 13. If you're unsure what
7 |  pin the on-board LED is connected to on your Arduino model, check
8 |  the documentation at http://www.arduino.cc
9 |
10 |  This example code is in the public domain.
11 |
12 |  modified 8 May 2014
13 |  by Scott Fitzgerald
14 |  */
15 |
16 |
17 | // the setup function runs once when you press reset or power the board
18 | void setup() {
19 |   // initialize digital pin 13 as an output.
20 |   pinMode(13, OUTPUT);
21 | }
22 |
23 | // the loop function runs over and over again forever.
24 | void loop() {
25 |   digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage level)
26 |   delay(1000);           // wait for a second
27 |   digitalWrite(13, LOW); // turn the LED off by making the voltage LOW
28 |   delay(1000);           // wait for a second
29 | }
```

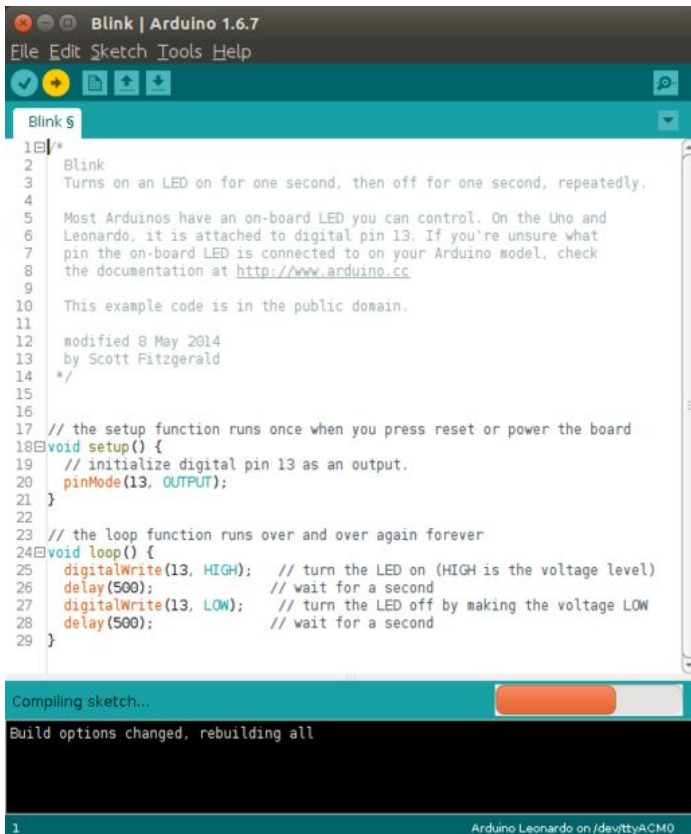
## Step 5: Uploading a program to the Arduino

As an example, we'll upload the Blink example to the Arduino, but since it's already installed by default, change the value of the delay functions to 500 instead of 1000, otherwise we won't see any difference.

Now connect your Arduino to your computer, if you haven't already. Wait for your computer to recognize the new USB device, and go to the Tools > Board menu in the Arduino IDE, and select your board. Then in Tools > Port, select the right port. In Windows, this will probably be a COM port. If there are multiple ports available, unplug your Arduino, then plug it back in, and see which of the ports disappears and reappears, to know the port of your Arduino. On linux, it's probably /dev/ttyACM0 (/dev/ttyS0 is the serial port on your motherboard, so that's not the right one).

In the bottom right corner, you should now see the name of your board, and the port it's connected to.

Now click the right-pointing arrow in the top left corner to upload Blink to the board. You could also use the shortcut CTRL+U. If all goes right, it should start compiling, and then upload. While it uploads the TX and RX lights should flash. When it's complete, your led on pin 13 should now blink twice as fast.



```
1 Blink
2 Turns on an LED on for one second, then off for one second, repeatedly.
3
4 Most Arduinos have an on-board LED you can control. On the Uno and
5 Leonardo, it is attached to digital pin 13. If you're unsure what
6 pin the on-board LED is connected to on your Arduino model, check
7 the documentation at http://www.arduino.cc
8
9 This example code is in the public domain.
10
11 modified 8 May 2014
12 by Scott Fitzgerald
13
14 */
15
16
17 // the setup function runs once when you press reset or power the board
18 void setup() {
19   // initialize digital pin 13 as an output.
20   pinMode(13, OUTPUT);
21 }
22
23 // the loop function runs over and over again forever
24 void loop() {
25   digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage level)
26   delay(500); // wait for a second
27   digitalWrite(13, LOW); // turn the LED off by making the voltage LOW
28   delay(500); // wait for a second
29 }
```

Compiling sketch...  
Build options changed, rebuilding all

1 Arduino Leonardo on /dev/ttyACM0



## Step 6: Driving more LEDs

Well, let's just face it: flashing one little LED is not that impressive... So in this step, I'll show how you can drive more LEDs.

### Calculating the resistor values for our LEDs

Before you begin, we'll have to calculate the resistor to use in series with our LED. The formula we'll need, is **Ohm's Law**:  $R = V/I$

But the problem here is that we don't know the voltage and the current of the resistor yet... Let's find it!

We connect the resistor in series with our LED, and we know that the voltage of the Arduino's output pin is 5v, so if we add the voltage of the LED to the voltage of the resistor, this should equal 5v. Therefore:  $V_R + V_L = 5v$ , and  $V_R = 5v - V_L$ .

Since they are in series, this also means that all electrons that pass through the LED, will go through the resistor as well. Since current is defined as the amount of electrons per unit of time, we know that the current through the resistor is equal to the current through the LED. Therefore:  $I_R = I_L$ .

Now we can use Ohm's Law to find the resistance of our resistor:  $R = V_R/I_R = (5v - V_L) / I_L$

(image 1)

If you are using a 3.3v Arduino, you can just substitute the 5v with 3.3v. This is just the supply voltage, it can be anything.

The values for  $V_L$  and  $I_L$  can be found in your LED's datasheet, or you can use the table below.

$I_L$  should be less than 20mA, otherwise, you'll kill your LED. But most of tss.

Note that you have to divide this value (in milliamps) by 1000, to get Amps, the SI unit of current. Otherwise, you won't get your answer in Ohms (but in kilo-ohms instead).

$V_L$  is the LED's forward voltage drop. (LEDs that emit lower energy light, with longer wavelengths, like red and infrared have a much lower voltage drop than light with

higher-energy photons, like blue or ultraviolet light.)

Color	Forward voltage (V <sub>L</sub> )
Red	1.7v - 2.2v
Orange	2.0v
Yellow	2.1v
Green	2.2v
Blue	3.2v - 3.8v
White	3.2v - 3.8v

This is just an approximation, though, it will not hold true for every LED.

You probably won't have a resistor of the exact value you just calculated, so round up to the next value in the E-12 series. This series only has the values 1.0, 1.2, 1.5, 1.8, 2.2, 2.7, 3.3, 3.9, 4.7, 5.6, 6.8, 8.2, and those values multiplied by a power of ten, like 10, 100, 1,000, 10,000 etc.

For example, my result was 65?, so I took a resistor of 68?.

It is better to take a resistor with a higher value, so you don't kill your LED.

## Connecting the LEDs to the Arduino

Like I explained earlier, an LED is basically a diode, so it lets the current pass in only one direction. If we connect the LED the other way around, nothing will happen.

An LED has a cathode and an anode: in schematic representation, the direction of the arrow is the conventional current flow direction, so it points to the negative side: the cathode.

When you take a close look at an LED, you'll see that one side is flat, and the other is round. Also, one leg is shorter than the other: the flat side, with the shorter leg is the cathode, and we'll connect this side to the negative, or ground of the Arduino.

It doesn't matter where you put the resistor, as long as it's in series with the LED, so you can connect it either to the cathode or to the anode.

If we know that only one LED will be on at the same time, we can use one resistor for all our LEDs.

Take a look at images 2-3-4, then connect the resistor to the ground and one of the horizontal rails of your breadboard, and an LED with the short leg into the same rail, and the long leg in pin 12 on the Arduino.

Note that you have to calculate the specific resistor for your LED. You can use the formula above, or use an online tool, or an app (I use [this tool](#) and [this app](#)). If you click the question marks next to the fields, the tool will give you some extra info, and the app has a drop-down menu for selecting the color.

If you are in doubt, or - like me - just too lazy to look it up and find the right resistor, you can just use any resistor between 220? and 330?. This only holds true if you use it with a 5v Arduino! 270? and 330? are a bit high to use with a 3v3 Arduino, in that case, use something between 100? and 220?. If you want to use an LED with a higher voltage, you'll need to calculate it.

When you have connected the LED and resistor to the Arduino, you can replace pin 13 in the Blink example by 12. You can use the shortcut CTRL+F: in the 'Find' field, enter 13, and in the 'Replace with' field, enter 12, then click 'Replace All'. Now upload the sketch to your Arduino: the LED you've attached should now blink.

Still not impressed? No? Well, you're right...

But now you've got the basics, we can do something more spectacular...

## More LEDs

Connect 5 extra LEDs, to pins 2, 4, 6, 8 & 10 with their longer leg, and connect their shorter leg to the same rail on the breadboard as the LED on pin 12 and the resistor.

Now download the zip file with the other examples, extract it, and open Blink6-a in the Arduino IDE.

If you remember the Blink example, it shouldn't be too hard to predict what this program does. Upload it to your Arduino, and check if your hypothesis was correct!

## Optimizing the code

If you take a closer look, you'll see that some pieces of the code are just 6 times the same commands, with just a different pin number... Well, there's a way to solve this: we call it the **for-loop**.

Open the Blink6-b sketch from the zip file. When you upload it to the Arduino, you'll see that it does exactly the same as Blink6-a, but it has much fewer lines of code.

This is all thanks to the two for-loops that are used.

But before I can explain the for-loop, I'll first have to explain how variables work:

On the first line, there's a new expression 'int delayTime = 200;'. This is the so-called declaration of the variable 'delayTime'. A variable is just a piece of memory, where you can store pretty much anything you like. A variable has a **data type**, a name, and a value: in this case the data type is 'int': an integer, a whole number and can be positive or negative. The name is 'delayTime', this is just an arbitrary name you give the variable, it can have letters and numbers in it, and some symbols like an underscore. If the name consists of more than one word, capital letters are used for every new word, because you can't use spaces. Choosing appropriate names for your variables will make a huge difference in the readability of your code. With 'int delayTime;', the variable is correctly **declared**, but doesn't have a value yet. Assigning an initial value to the variable is called **initializing**. In this case, we initialize variable delayTime to 200. Now every time we use 'delayTime' in our code, it will be replaced by the number 200. If you read a variable without first initializing it, it will just give you 0, in the best case, but probably just a random value that happened to be in that place of the memory, possibly causing errors in your program.

You also have to know that a variable has a so-called **scope**. In this example, delayTime is declared at the top, so it is a **global variable**. This means that you'll be able to access it inside every function, every loop, in the whole program. A variable can also be **local**, meaning you can only use it in the scope in which it is defined. For example, if we were to move the first line to the setup, we would get the error

'delayTime' was not declared in this scope

<http://www.instructables.com/id/A-Beginners-Guide-to-Arduino/>

when we'd try to access it in the main loop (`delay(delayTime);`), because this is a different scope than `setup`. But local variables can come in handy, when you don't have lots of memory to spare, for example, since the local are 'deleted' when you leave the scope they are declared in.

Most of the time, the boundaries of a scope are curved brackets.

But why use this variable? Well, instead of having to change all the `delay(200);` functions, we can just change the first line. In this case it's not very useful, but if you have really large program, it really matters.

Arduino reference: variables

Arduino reference: int

Arduino reference: variable scope

Another useful thing about variables is that you can change them, while your code runs. And this is exactly what happens in a for-loop. The first part between the brackets is just the declaration of a certain variable 'i', and we initialize it to 2. The next part is 'i <= 12', this means that the for-loop will repeat as long as i is less than or equal to 12. The third and last part 'i = i+2' means that every time the loop is repeated, the value of i will be increased with 2.

All this means that it will first execute the code inside the brackets with a value of 2 for i, sets pin 2 as an output. Now it has reached the end of the code, so 2 is added to i ? i is now equal to 4, and the code runs again with this new value for i: pin 4 is set as an output. 2 is added again, i is now equal to 6, the code is run again, 2 is added ... and so on. Until i equals 12. When the loop gets executed for i = 12, everything is normal, but when it reaches the end of the code now, and 2 is added again, i doesn't meet the condition 'i ? 12' (i is now 14). The program exits the for-loop. Every pin with an LED attached is now set as an output, with only two lines of code, instead of six.

Arduino reference: for-loop

## Summary:

Finding the right resistor for your LED

1. Look up the forward voltage in the LED's datasheet, or in the table above. This is  $V_L$
2. Look up the forward current in the LED's datasheet, or use something between 2 and 20mA, and convert this number to Amps. This is  $I_L$
3. Find the voltage of your supply, 5v or 3.3v for most micro-controllers like Arduino. This is  $V_S$
4. Now use this formula to find the resistor value:  $R = (V_S - V_L) / I_L$
5. Round this value up to the closest resistor you have. You can always use a higher value than the resistance you calculated, but if you use a lower resistance, this will drastically shorten the lifespan of your LED. If the resistance is too high, the LED will be very dim.
  - The round side of the LED, with the longer leg, is the anode, and is the positive side.
  - The flat side of the LED, with the shorter leg, is the cathode, and is the negative side.
  - CTRL+F opens the search/find/replace window in the Arduino IDE
  - `int delayTime;` this is the declaration of a variable of data type int, called 'delayTime'
  - `int delayTime = 200;` this is the declaration of a variable of data type int, called 'delayTime', and its initialization to a value of 200.
  - Global variables can be accessed everywhere in the program, local variables only in the scope where they were declared.
  - `for(int i = 0; i < 10; i++) { }` is a for-loop that initializes variable i to a value of 0, adds 1 to i every time it repeats, and repeats as long as i < 10 (10 times).

## Extra: Calculating the power dissipation in the resistor

If you use a 5v power supply, like the one on the Arduino, a normal ¼ watt resistor will be enough, but when you use a higher supply voltage, or an LED that draws more current, the power dissipation in the resistor will be too high, and it will burn out.

Therefore, we need to calculate what the power dissipation in the resistor is.

(Image 6)

The formula to calculate power is  $P = V \cdot I$ . So we can just calculate the voltage across the resistor: the supply voltage minus the voltage across the LED:  $V_R = V_S - V_L$ .

And the current through the resistor is the same as the current through the LED:  $I_R = I_L$ .

Therefore:  $P_R = (V_S - V_L) \cdot I_L$

In the example shown in image 6, the power dissipation in the resistor is very low, so a ¼ watt resistor will be sufficient, but if the calculated value is higher, you'll need a more powerful (= thicker) resistor.

These values are, however, just a theoretical assumption.

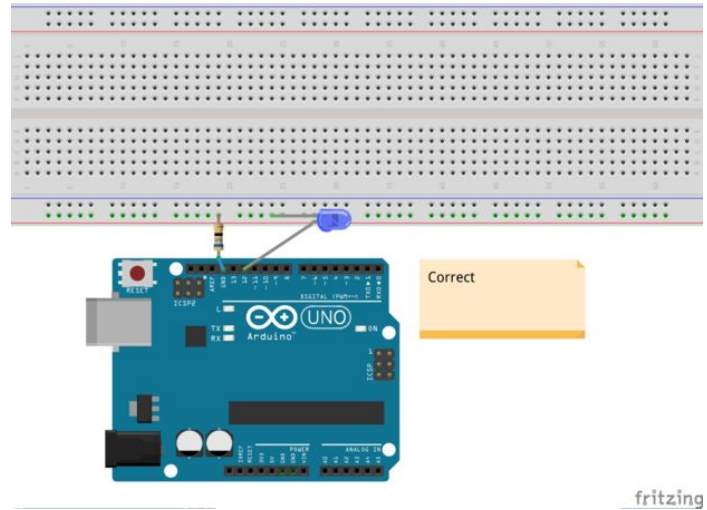
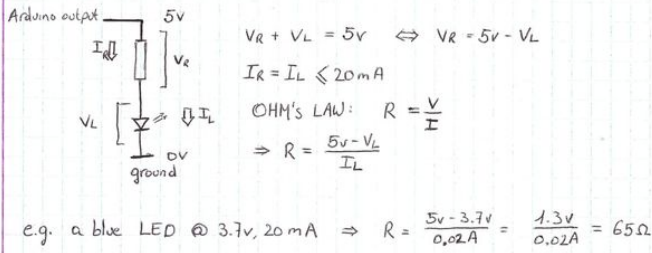
If you really want to know the actual power the resistor dissipates, you'll have to measure the current or the voltage across it, or in the best case, both.

In the case you can measure both the current and the voltage (case A), you can just use the same formula.

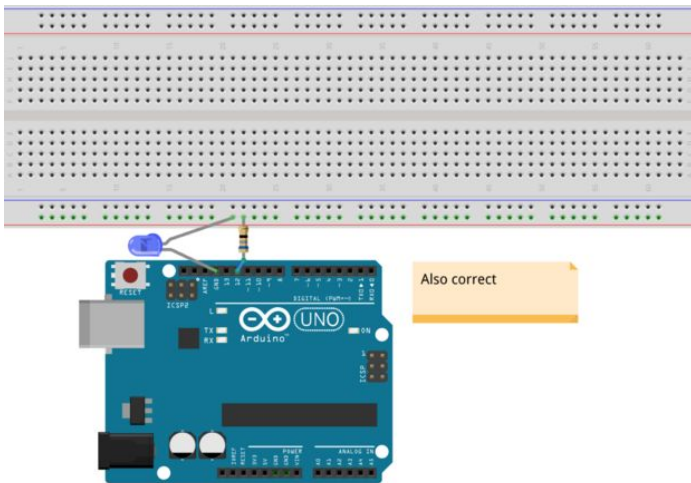
If you can, however, only measure one of these values, you can use Ohm's Law to substitute either the voltage or the current in the formula for power. (case B & C)

Only case A gives you the real power dissipation, in case B & C, you'll need the value of the resistor. The value written on the resistor has a certain tolerance, so you'll have some deviation in your wattage as well.

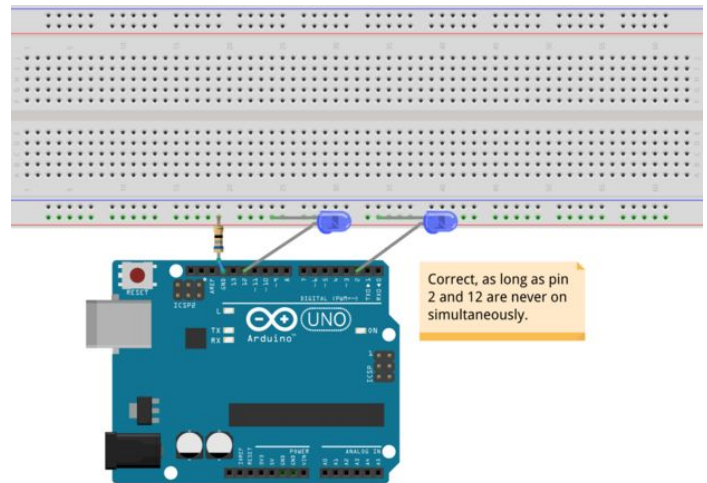
## Calculating an LED resistor



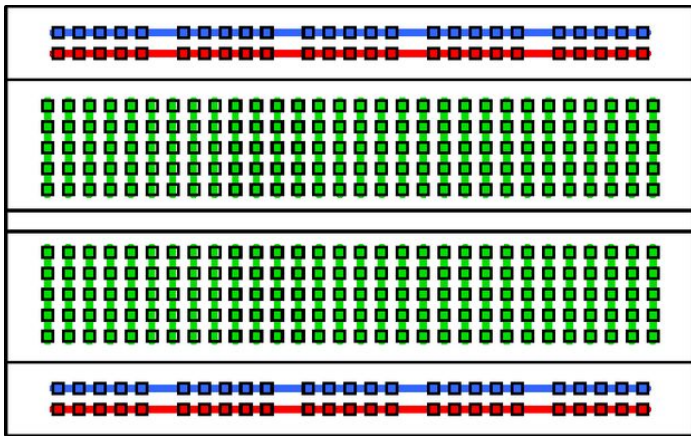
fritzing



fritzing



fritzing



## Calculating power dissipation in the resistor (theoretical)

Formula for power:  $P = V \cdot I$   
 $\Rightarrow P_R = V_R \cdot I_R = 1.3V \cdot 0.02A = 0.026W$

## Calculating power dissipation in the resistor (in practice)

Case A: Voltage and current can be measured

$P_R = V_R \cdot I_R = 1.13V \cdot 0.0164A = 0.0185W$

Case B: only voltage can be measured

OHM'S LAW:  $I = \frac{V}{R} \Rightarrow P_R = V_R \cdot \frac{V_R}{R} = \frac{V_R^2}{R} = \frac{(1.13V)^2}{68\Omega} = 0.0188W$

Case C: only current can be measured

OHM'S LAW:  $V = I \cdot R \Rightarrow P_R = \frac{V_R}{I_R} \cdot I_R = I_R^2 \cdot R = (0.0164A)^2 \cdot 68\Omega = 0.0183W$

## Step 7: Even more blinking LEDs Programming the LED sequence

Open the example blink7.

On line 3, you'll see a new structure: we call this an **array**. An array is basically a list of variables, in this case, it is a list of variables of the type int. We use square brackets to indicate that it is an array. To initialize the array, we use curved brackets, and the values are separated by commas.

When we want to use a value from an array, we'll need to specify which place in the array we want the value of. Let's use the array in blink7 as an example :

```
int array[] = {1,2,3,4,5,6,5,4,3,2};
```

if we want to get the first value of the array, we use

```
array[0]
```

<http://www.instructables.com/id/A-Beginners-Guide-to-Arduino/>



The zero between square brackets is called the **index**. Arrays are **zero-based**, this means that the first element in the array has index zero. This can be confusing at first, for example, `array[5]` will return 6, where you'd expect it to return 5.

You can imagine that this can cause a lot of problems... That's why this error has its own name: an **off-by-one error**, or **OBOE** for short.

When you try to read a value that isn't inside the array, `array[10]` for example, the Arduino will just read the next place in the memory, without realizing that the value it's reading isn't a part of the array anymore. It will just give you the value it finds at that particular spot in its memory.

But things get even worse when you are writing to an index outside of the array, because you may be overwriting other data, like variables or pieces of code that happen to be at that place in memory! A bad idea ...

If you want to declare an array without initializing it yet, you can use

```
array[10];
```

Note that 10 is the number of elements, while the last element will be `array[9]` !

If you initialize it on the same line, like in `blink7`, the Arduino IDE will count the elements for you, and you don't have to specify anything between the brackets.

If you want to know the number of elements in the array, you can use the **`sizeof(array)`** function. This function returns the size of the array, in bytes (1 byte = 8 bits). This however, isn't the amount of elements, in `blink7`, `sizeof(array)` will return 20. This is because every element is an int, and an int is 2 bytes long. So the total number of elements is  $20/2 = 10$ .

We use this to exit our for-loop when we've read the whole array: As long as the index is less than the number of elements, we can safely read the array. We start with `i = 0`, since the first element has index 0. This time we only increase `i` with 1. The notation `'i++'` is exactly the same as writing `'i = i + 1'`, it's just shorter. Another way of writing this would be `'i += 1'`.

Since LED #1 is connected to pin 2, LED #2 to pin 4, 3 to 6 etc, we multiply the LED number by 2 to get the pin number.

Arduino reference: Array

Arduino reference: Sizeof

Arduino reference: Increment

Arduino reference: Compound Addition

You can change the values in the array to make your own sequence, and you could even add or delete elements, since we made our program independent of the length of the array, by using the `sizeof()` function.

## Summary:

- An array is a list of values (or characters, or other arrays) and uses square brackets []
- This is the declaration of an array: `'int array[10];'`
- This is the declaration and initialization of an array: `'int array[] = {1,2,3,4,5,6,5,4,3,2};'`
- Array indices are zero-based, meaning the first element has index 0
- `sizeof(array)` returns the size in bytes of the array
- `sizeof(data type)` returns the size in bytes of the data type
- `sizeof(array)/sizeof(data type of array)` gives you the number of elements in the array

## Extra: 2-dimensional arrays (Matrices)

In an array, the elements can not only be data types, like ints, but also other arrays. This allows you to have so-called 2-dimensional arrays, that can be compared to **matrices**.

This is used in example `blink8`: now you can set both the led number and the delay in the array.

```
int array[][2] = {
  {1, 600},
  {2, 100},
  {3, 100},
  {4, 100},
  {5, 100},
  {6, 600},
  {5, 100},
  {4, 100},
  {3, 100},
  {2, 100}
};
```

Note that we have to specify all dimensions except the first one between the square brackets, when declaring a multi-dimensional array.

The syntax to get a value out of the array is

```
array[row][column]
```

Open the example `matrix_sum`. This example won't do anything if you upload it, it doesn't have any outputs, it's just for learning purposes.

It creates 4 matrices (2D arrays), two of which have values. Then it just calculates the sum of the two matrices, and the transposed matrix for the first one.

This is done by using nested for-loops. You can follow this explanation on the image above.

The numbers are the order the for-loop will go over them. `i` is used for the rows, `j` for the columns. They start at the first column and the first row (1,1) in matrix notation, `[0][0]` for array notation, then the column is incremented (1,2) or `[1][0]`, the column is incremented again (1,3) or `[2][0]` when `j` is incremented again, the `j`-loop exits, because `j` is no longer less than three. `i` is incremented, `j` resets to zero: (2,1) or `[0][1]`, `j` is incremented: (2,2) or `[1][1]`, then (2,3) or `[2][1]`. `i` is incremented, `j` resets to zero: (3,1) or `[2][0]`, then (3,2) or `[2][1]`, and finally (3,3) or `[2,2]`. Now `j` exits, and `i` exits as well.

The calculation of the transposed matrix is similar, it just swaps the columns and the rows:

```
transposeMatrix[i][j] = matrixA[j][i];
```

Inside these arrays, you can use more arrays, basically creating multi-dimensional spaces. In theory, the number of dimensions is unlimited, but once you use more than three dimensions, it gets really complicated, so other methods are preferred.

index	array[0]	array[1]	array[2]	array[3]	array[4]	array[5]	array[6]	array[7]	array[8]	array[9]
value	1	2	3	4	5	6	5	4	3	2

10 elements

```

blink7 | Arduino 1.6.7
File Edit Sketch Tools Help

blink7
1 int delayTime = 200; // declare a variable of data type int, called delayTime, and initialize it to 200
2
3 int array[] = {1,2,3,4,5,6,5,4,3,2}; // declare a new array, containing the LED number to turn on.
4 // the setup function runs once when you press reset or power the board
5 void setup() {
6 // initialize digital pins 2, 4, 6, 8, 10 and 12 as outputs.
7 for (int i = 2; i <= 12; i = i + 2) { //make a variable of type int (integer) called i,
8 // that starts from 2, repeats as long as i is less
9 // than or equal to 12, and every time it repeats,
10 // we add two to i.
11 pinMode(i, OUTPUT);
12 }
13 }
14
15 // the loop function runs over and over again forever
16 void loop() {
17 for (int i = 0; i < sizeof(array) / sizeof(int); i++) {
18 digitalWrite(array[i] * 2, HIGH); // turn the LED on the pin defined in the array on
19 delay(delayTime); // wait for delayTime ms
20 digitalWrite(array[i] * 2, LOW); // turn the LED on the pin defined in the array off
21 }
22 }
    
```

Done Saving.

Arduino Leonardo on /dev/ttyACM0

```

blink8 | Arduino 1.6.7
File Edit Sketch Tools Help

blink8
1 int array[][2] = {
2 {1, 600},
3 {2, 100},
4 {3, 100},
5 {4, 100},
6 {5, 100},
7 {6, 600},
8 {5, 100},
9 {4, 100},
10 {3, 100},
11 {2, 100}
12 }; // declare a new array, containing the LED number to turn on, and the time it will be on.
13
14 // the setup function runs once when you press reset or power the board
15 void setup() {
16 // initialize digital pins 2, 4, 6, 8, 10 and 12 as outputs.
17 for (int i = 2; i <= 12; i = i + 2) { //make a variable of type int (integer) called i,
18 //that starts from 2, repeats as long as i is less than or equal to 12, and every time
19 //it repeats, we add two to i.
20 pinMode(i, OUTPUT);
21 }
22 }
23
24 // the loop function runs over and over again forever
25 void loop() {
26 for (int i = 0; i < sizeof(array) / sizeof(int) / 2; i++) {
27 digitalWrite(array[i][0] * 2, HIGH); // turn on the LED on the pin defined
28 // in the first column of the array (column index 0)
29 delay(array[i][1]); // wait for the time specified in the second column of
30 // the array (column index 1)
31 digitalWrite(array[i][0] * 2, LOW); // turn the LED on the pin defined in the array off
32 }
33 }
    
```

Done Saving.

Sketch uses 4,530 bytes (15%) of program storage space. Maximum is 28,672 bytes.  
Global variables use 188 bytes (7%) of dynamic memory, leaving 2,372 bytes for local variables. Maximum is 2

Arduino Leonardo on /dev/ttyACM0

```

matrix_sum | Arduino 1.6.7
File Edit Sketch Tools Help

matrix_sum
1 /*
2 * This is just an example, it won't do anything if you upload it to the Arduino.
3 */
4 int matrixA[][3] = {
5 {1, 2, 3},
6 {4, 5, 6},
7 {7, 8, 9}
8 };
9
10 int matrixB[][3] = {
11 {9, 8, 7},
12 {6, 5, 4},
13 {3, 2, 1}
14 };
15
16 int sumMatrix[3][3]; // the sum of matrix A & B
17
18 int transposeMatrix[3][3]; // the transposed matrix of matrix A
19
20 void setup() {
21 }
22
23 void loop() {
24 for (int i = 0; i < 3; i++) { // two (nested) for-loops, to go over the rows (=i) and columns (=j)
25 for (int j = 0; j < 3; j++) {
26 sumMatrix[i][j] = matrixA[i][j] + matrixB[i][j];
27 }
28 }
29
30 for (int i = 0; i < 3; i++) {
31 for (int j = 0; j < 3; j++) {
32 transposeMatrix[j][i] = matrixA[i][j]; // note that the indices are swapped
33 }
34 }
35 delay(10000);
36 }
    
```

Done Saving.

Sketch uses 4,122 bytes (14%) of program storage space. Maximum is 28,672 bytes.  
Global variables use 220 bytes (8%) of dynamic memory, leaving 2,340 bytes for local variables. Maximum is

Arduino Leonardo on /dev/ttyACM0

<b>matrixA</b>	<b>j = 0</b>	<b>j = 1</b>	<b>j = 2</b>
<b>i = 0</b>	1	2	3
<b>i = 1</b>	4	5	6
<b>i = 2</b>	7	8	9

## Step 8: Input from a button

Of course, you can use the I/O pins of the Arduino as inputs as well. In this step, we'll just use push buttons as input devices, but of course, you can use any switch.

### Pull-up and pull-down

The Arduino works with logical inputs: 1 = 5v, 0 = 0v. To make our button output these voltages, we'll use a pull-up or a pull-down resistor. (image 1 & 2)

In the case of a pull-down resistor (image 1), we connect one leg of the switch to 5v, and the other leg through a resistor (10k? in this case) to ground (0v). The leg with the resistor connected goes to the input pin on the Arduino.

This way, when the button is not pressed (and doesn't connect the 2 legs), the input is at 0v, because it's connected to ground through the resistor. When you press the button, the input is at 5v, because it's directly connected to 5v through the switch. The resistor doesn't matter when you press the button, it just makes sure that the input is at 0v when the button is not pressed.

A pull-up resistor (image 2) works in exactly the same way, but everything is swapped: the first leg is connected to ground, instead of 5v, the second is connected to 5v, through a resistor (hence the name pull-up resistor, since it pulls it up to 5v). The input pin still connects to the side with the resistor, but now, it is high when the button is not pressed, and goes low when the switch is closed.

Button	Logical state pull-up	Logical state pull-down
released	1	0
pressed	0	1

Now, connect a push button with a pull-down resistor to pin 3 of the Arduino, and a push button with a pull-up resistor to pin 5. Then connect two LEDs (with their appropriate resistor) to pins 10 and 12. (image 3 & 4)

Now open example button2, and open it. This program just reads the two inputs, and sets the outputs in the same state.

There are only two new things, and they are really obvious: instead of the OUTPUT constant, we use INPUT to set the pins of our buttons as inputs, and the digitalRead(pin) function just returns the state of the given input pin.

Note: using pinMode(pin, INPUT) is actually unnecessary, since all pins on the Arduino are inputs by default, but it's often done anyway, to make the code more readable.

When you upload the sketch, press the buttons, and you'll see that the table above is indeed correct: the LED on pin 12 is always on, until you press the button on pin 5, this is because it has a pull-up resistor.

If you want the LEDs to light up only when you push the button, you can use the Boolean not-operator: this just changes a 'true' into a 'false' (or a 1 into a 0) and vice versa. In C++ (in the Arduino IDE), this operator is an exclamation mark (!)

```
digitalWrite(12, !digitalRead(5));
```

### Internal pull-up resistors

It would be really inconvenient if we had to use an extra resistor and an extra piece of wire, every time we want to use a normal switch. That's why the chip on the Arduino has a built-in pull-up resistor on every pin.

There are two ways to enable them:

```
pinMode(pin, INPUT);  
digitalWrite(pin, HIGH);
```

```
pinMode(pin, INPUT_PULLUP);
```

Both have the same effect, but the latter one is preferred, because it is more readable.

Note: if you forget to use pinMode(pin, OUTPUT), and you use digitalWrite(pin, HIGH) afterwards, you'll just enable the pull-up resistor, because all pins are set as inputs by default.

Now connect the push buttons without the resistors, just the connection to the ground (as shown in image 5)

You can see that we don't need to use the 5v pin of the Arduino anymore, and if we were to produce this at a large scale, those two resistors we saved would make a significant difference in production cost.

Open the example button2-b. As you can see, I used the two ways to enable the pull-up resistors. Also note that I used the 'not' operator, so the LEDs are on when the button is pressed.

### Summary

- To use buttons and switches with your Arduino, you have to use a pull-up or pull-down resistor.
- pinMode(pin, INPUT\_PULLUP); enables the internal pull-up resistors of the Arduino.
- digitalWrite(pin, HIGH); on an input pin has the same result.
- digitalRead(pin) returns the state of the input pin 1 = 5v, 0 = 0v.  
If you use a button with a pull-up resistor (e.g. the internal one), 1 means that the button is not pressed, 0 means it's pressed.
- You can use the not-operator (!) to swap 1 and 0. E.g. !digitalRead(pin) returns 0 when the button is not pressed, and 1 when the button is pressed.

### Extra: Direct port manipulation

DigitalRead, digitalWrite and pinMode are great and simple functions, but they are relatively slow. Also, you can't turn on 2 pins on or off at exactly the same time, and writing 8 bits simultaneously for parallel communication is not possible either. Sometimes, when you're running short on memory, these 3 functions can use a lot of the available space, too.

The solution to these problems is **direct port manipulation**. The Atmel chip has some (3 on most Arduinos) **registers** for the I/O pins, these are just bytes that store the info on whether a pin is an input or an output, whether it is set high or low, etc. Every bit of these bytes corresponds to an I/O pin on the Arduino.

On the Arduino Uno, port D contains pins 0 to 7, port B pins 8 to 13, and port C A0 to A5.

<http://www.instructables.com/id/A-Beginners-Guide-to-Arduino/>

There are 3 registers to control the I/O (where x is the port letter):

- **DDRx**: Data Direction Register: this sets whether the pins of the port are inputs(1) or outputs (0). (pinMode)
- **PORTx**: Port Data Register: This is to set outputs high or low, and disable or enable the input pull-up resistors. (digitalWrite)
- **PINx**: Port Input Register: This byte contains the state of the digital inputs. If the pin is an output, it will just give you the output state.

In the image above, you can see the entire **pin mapping** of the Arduino Uno, the port numbers are in the yellow fields next to the pins. (image credit)

Since every bit of the byte represents one pin, it is easier to write the values in **binary notation**. You can do this by adding a capital **B** before the number, for example, B111 is 7 in decimal ( $2^2 + 2^1 + 2^0$ ).

Similarly, you can use a leading **0** to use **octal notation**, or **0x** for **hexadecimal notation**, however in this case using these two notations doesn't really make sense.

When counting bits, the rightmost (least significant, LSB) bit is bit 0, so it corresponds to the first pin of the port, while the MSB (most significant bit) corresponds to the eighth pin of the port.

Some examples:

Setting pin 7 to an output, and pins 0-6 as inputs:

```
DDRD = B10000000;
```

Setting (output) pin 7 high:

```
PORTD = B10000000;
```

Enabling the internal pull-up resistor on (input) pin 6:

```
PORTD = B01000000;
```

Reading the state of pins 0 to 7:

```
byte state = PIND;
```

However, using it like this can cause some problems: e.g. in the second example, pin 7 is set high, but all other pins in the port are set to zero, regardless of their previous state. To change only one pin at a time, we can use some bitwise operators.

To set one bit high, without changing the other bits, we can use the bitwise **or-operator** (`|`). Note: this is only one `|`, whereas the boolean or-operator is `||`. Bitwise means that it is applied to every bit separately. We use a mask to set the right bit high: the bit we want to set high is 1, and all other bits are 0.

```
byte previousPORTD = PORTD; // read the data register, and store it in a variable
PORTD = previousPORTD | B10000000; // set bit seven high
```

If a bit in the mask is one, this bit will be set to 1 in the PORTD register, if it is zero, it will just keep the value in previousPORTD. You can check out the truth tables and some examples in the images above.

To set one bit low, without changing the other bits, we can use the bitwise **and-operator** (`&`). Note: this is only one `&`, whereas the boolean and-operator is `&&`. Now we have to invert our mask: the bit we want to set low is zero, and all the other bits are one.

```
byte previousPORTD = PORTD; // read the data register, and store it in a variable
PORTD = previousPORTD & B01111111; // set bit seven low
```

This notation works just fine, but we can make it a little more readable. Instead of writing a whole binary number for our mask, we can use the left bitshift operator (`<<`). It just shifts all digits to the left for a given number of places. For example:  $1 \ll 2 = B100$ ,  $B101 \ll 1 = B1010$  and so on. This is also an easy way to calculate powers of two:  $1 \ll 7 = B10000000 = 128 = 2^7$

But we're going to use it to create our mask: for example, the mask B10000000 can be written as  $1 \ll 7$ , now you can easily see that it is the eighth pin in the register, without having to count the zeroes.

To create the inverse mask, for the and notation, we can use the bitwise **not-operator** (`~`) to invert every bit: B01111111 can be written as  $\sim(1 \ll 7)$ . (see image)

If you just want to flip one or more bits, you can use the **exclusive or-operator**. (`xor`, `^`). This returns 1 if one of the input bits is one, and 0 if both inputs are the same.

```
byte previousPORTD = PORTD; // read the data register, and store it in a variable
PORTD = previousPORTD ^ B10000000; // flip bit seven
```

We can use **compound operators** to shrink the code: `x = x + 1`; can be written as `x += 1`; for example. The same goes for bitwise operators. We won't need the temporary variable anymore.

```
PORTD |= 1<<7; // set bit 7 high
```

```
PORTD & ~(1<<7); // set bit 7 low
```

```
PORTD ^= 1<<7; // flip bit 7
```

To get one specific bit from the input register, for example, you could use the **bitRead(byte, bit)** function:

```
boolean state = bitRead(PIND, 6); // read the state of the 6th pin of port D
```

you could use some basic math to achieve the same result:

```
boolean state = (PIND>>6)%2; // read the state of the 6th pin of port D
```

This uses a **right bitshift** (`>>`) and a **modulo operator** (`%`): `>>` does the same as `<<`, but in the other direction. If PIND is B10101010, for example, `PIND>>6 = B10`, basically, it chops off the last 6 (binary) digits. The bit we wanted to check is now the rightmost bit. Modulo gives you the remainder of a division, e.g.  $10\%3 = 1$ , because  $3*3 + 1 = 10$ ,  $5\%6 = 5$ , because  $0*6 + 5 = 5$ ,  $23\%8 = 7$ , because  $2*8 + 7 = 23$ . `x%2` will give you a 0 if x is even, and 1 if x is odd. In binary notation, the last digit (rightmost bit) of an even number is 0, and the last digit of an odd number is 1. So if we just want to know the last digit of a binary number, we can just use `x%2`.

Another way to get only one bit of a number is using the **conditional operator** (`?:`):

<http://www.instructables.com/id/A-Beginners-Guide-to-Arduino/>

```
boolean state = (PIND &(1<<6)) == 0 ? 0 : 1; // read the state of the 6th pin of port D
```

PIND &(1<<6) will only keep the 6th bit of PIND, all other digits will be 0. If PIND is B10101010, for example, PIND & (1<<6) = B00000000, and if PIND is B01010101, PIND & (1<<6) = B01000000. From these examples, you can see that the result is zero if bit 6 was 0. So if we test if this result == 0, we know the state of bit 6. We use the conditional operator: condition ? resultIfTrue : resultIfFalse. If the condition is true, the operator will return the resultIfTrue, if it's false, it will return resultIfFalse.

**Note on compatibility:** the port-to-pin mapping depends on the chip. This means that you have to change your program if you want to use another Arduino. For personal use, this isn't so much of a problem, but if you're writing programs or libraries for the community or to share online, you should take this into account.

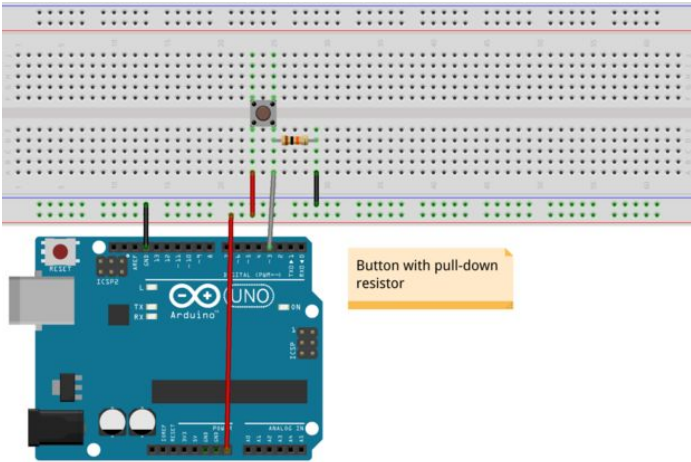
Arduino reference: Integer Constants

He killed my wire: direct port manipulation tutorial

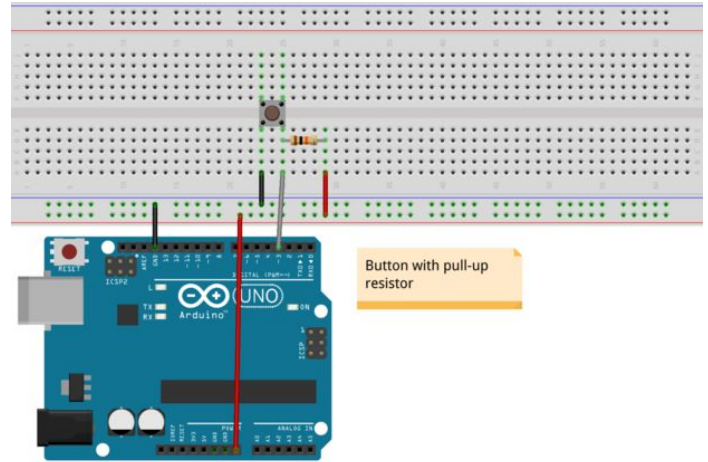
Arduino reference: Port Manipulation

The Atmel ATmega 328p datasheet p.91 14.4

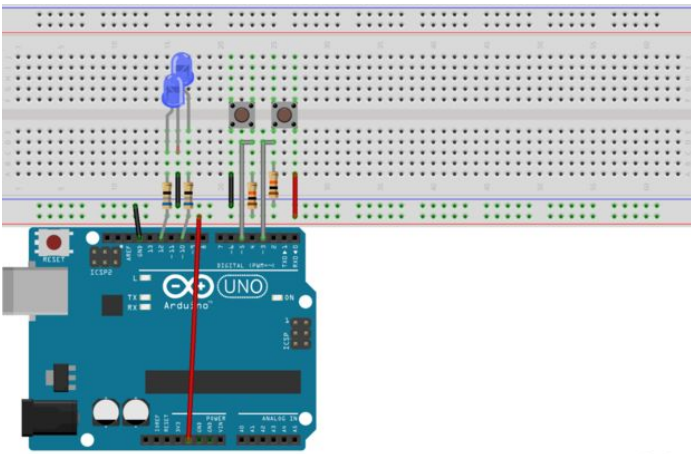
Arduino reference: bitRead



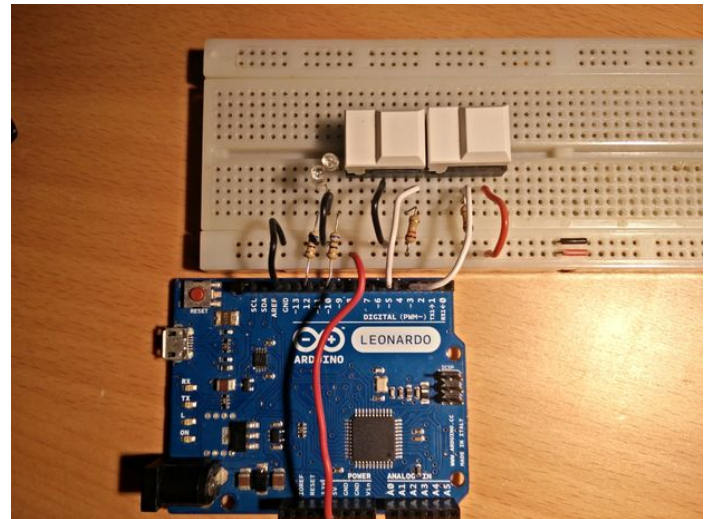
fritzing

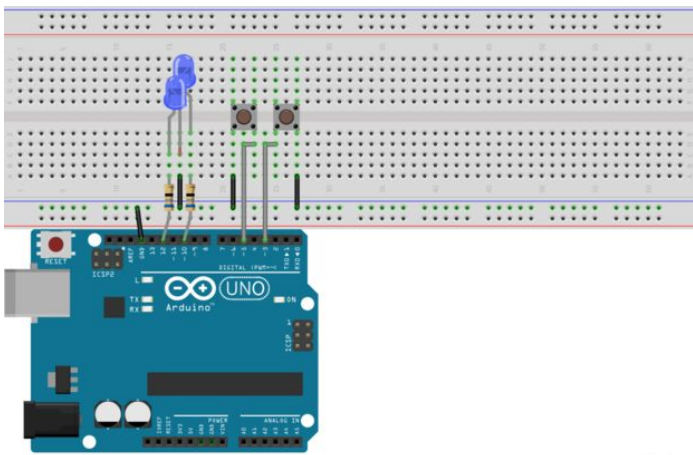


fritzing

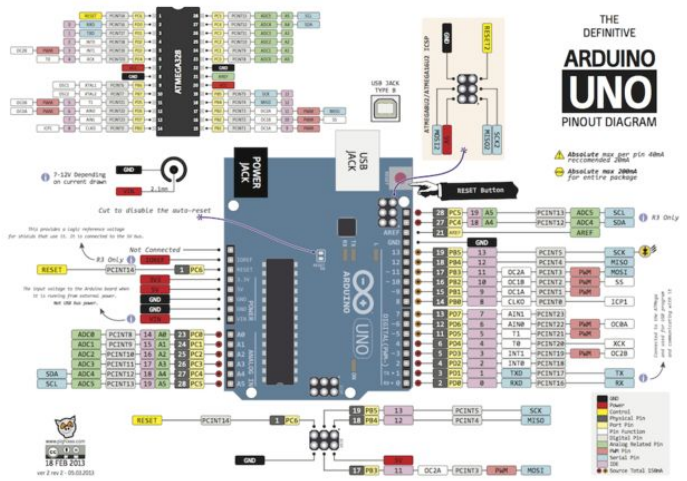


fritzing





fritzing



### Bitwise operators

Bitwise or		
A	B	A   B
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise exclusive or		
A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise and		
A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise not	
A	~A
0	1
1	0

x	1<<x	~(1<<x)
0	B1	B11111110
1	B10	B11111101
2	B100	B11111011
3	B1000	B11110111
4	B10000	B11101111
5	B100000	B11011111
6	B1000000	B10111111
7	B10000000	B01111111

0	0	0	0	0	0	0	0	= previousPORTD
or	1	0	0	0	0	0	0	
Setting bit 7 high	1	0	0	0	0	0	0	
	1	1	1	1	1	1	1	= previousPORTD
or	1	0	0	0	0	0	0	
	1	1	1	1	1	1	1	

1	1	1	1	1	1	1	1	= previousPORTD
and	0	1	1	1	1	1	1	
Setting bit 7 low	0	1	1	1	1	1	1	
	0	0	0	0	0	0	0	= previousPORTD
and	0	1	1	1	1	1	1	
	0	0	0	0	0	0	0	

0	0	0	0	0	0	0	0	= previousPORTD
xor	1	0	0	0	0	0	0	
Flipping bit 7	1	0	0	0	0	0	0	
	1	1	1	1	1	1	1	= previousPORTD
xor	1	0	0	0	0	0	0	
	0	1	1	1	1	1	1	

## Step 9: Communication with the computer

Up to this point, we only used the USB connection to upload new programs to the Arduino. But we can also use the connection to send data from and to the computer.

Some Arduino boards like the Leonardo have a microcontroller that handles the USB communication all by its own, but most boards have a second, smaller microcontroller, solely for the communication. On the Arduino Uno, this is the small black square between the USB connector and the TX-RX LEDs. The communication between this chip and the main microcontroller is done using a serial connection, then the second chip tells the computer 'Hey, I'm a serial port', and then simply converts the serial data from the main chip to USB format, and converts the messages from the computer to serial for the main microcontroller.

Serial means that bits are sent over the same wire, one after the other, whereas parallel connections send 8 bits or more simultaneously, on separate wires (like the large parallel printer port on the back of some older computers).

The Arduino uses two serial lines: one from the Arduino to the USB chip, the transmit line (TX), and one line from the USB chip to the Arduino, the receive line (RX). These letters are written next to pin 0 and 1. This means that these pins are in use when you have the connection to the computer running, so you can't use them for LEDs or buttons etc. If you stop the communication again, you can just use them as normal I/O pins.

If you want to connect the Arduino to another device using serial communication, you also have to use these two pins.

## Sending inputs to the computer

Open example serialButton, and upload it.

The breadboard configuration is the same as in the previous step.

Then open the **serial monitor**: Tools > Serial Monitor, click the magnifying glass in the top right corner or hit CTRL+SHIFT+M. Make sure autoscroll is enabled, and the baud is set to 9600.

You'll see a bunch of ones. Now press the push button connected to pin 3, and you'll see zeros. This is just the raw input from the pin.

Now open the **serial plotter**: Tools > Serial Plotter or CTRL+SHIFT+L. This will plot the values into a graph.

Let's take a look at the code:

In the setup, we add a new command: **Serial.begin(9600)**. This is just to start the communication, and the 9600 is the **baud rate**, the number of pulses per second, so the speed of the serial connection. 9600 is just the default value. If you set this to a different value, you'll have to change it in the serial monitor as well. Otherwise, it will be out of sync, and give you very strange characters. The opposite of **Serial.begin(...)** is **Serial.end()**. You can use this when you have to use pins 1 and 0 as normal I/O again, after using the serial connection, however it's not really recommended.

In the loop, you'll see the same **Serial** keyword, this time used with the **println(...)** function. This just prints out the value that's specified between the brackets, followed by a new line (**\n**).

(Note that this doesn't just convert it to binary to send it over serial, instead, it converts it to **ASCII**, and then sends it over serial, so that the computer can print it out. If you want to send binary bytes over serial, use the **Serial.write(...)** function.)

To get a new line, you could also use **Serial.print(...)** together with the **\n** (newline) character or the **\r** (carriage return) character, that indicate a line ending, instead of the **\n**.

```
Serial.print(digitalRead(3));  
Serial.print('\n');
```

The single quotation marks indicate that it is a character.

Another special character is the **\t** (tab), let's use it to print the inputs of both switches. This is example serial2Buttons. Upload it, and open the serial monitor to see the result.

Arduino reference: [Serial](#)

## Sending commands to the Arduino

The communication also works the other way around: let's write a program to control LEDs from the computer.

Open the example serialLEDs.

Upload it, open the serial monitor, and try sending values between 0 and 3. Now send 4. Get how it works?

Let's check out the code:

The setup should look quite familiar, except one command: this is a **while-loop**. A while-loop repeats, as long as the condition between the brackets is **true** (you could compare it to a for loop, but without the first and last argument, only the condition.) I explicitly used the curved brackets to indicate it is a loop, but you could also just use a semicolon instead: `while(!Serial);`

**Serial** returns true when the serial communication is active (when you open the serial monitor, for example). So if we add a not-operator (!), the expression is true when the communication is not active. So the while-loop keeps on repeating while the communication is not active, and essentially waits for it to become active. Then we know we can start sending data with the `println(...)` function. If we wouldn't use the while-loop to wait, some boards like the Leonardo (and other boards that have USB capabilities in the main chip) will just lose the first data. It does try to send it to the computer, but nothing there is listening for serial input.

In the loop, there's another new statement: the **if-statement**, this is probably the most important statement in computer science. It does exactly what the name implies: it executes a piece of code, only when a certain condition is true. Otherwise, the code between the curved brackets is ignored.

Now I'll take a moment to explain how the Arduino receives serial messages:

As I explained earlier, serial sends one bit after the other, byte by byte. This means that your message, "test", for example, gets split up in pieces of 1 byte. A character is one byte in size, so it will look something like 't' 'e' 's' 't' when you send it over serial. When the Arduino receives these bytes, it will store them in a **buffer** (just a small piece of memory to temporarily store it). It receives it byte by byte, so the buffer will look something like this "" (empty) "t", "te", "tes" "test".

When the buffer is empty, the **Serial.available()** function will return 0, if there's received data in the buffer, it will return the number of bytes in the buffer, so 4 in this case. If you call the **Serial.read()** function, it will read the first byte ('t') from the buffer, delete it from the buffer, and shift the buffer, so it now contains "est", and **Serial.available()** returns 3. If you call **Serial.read()** again, it will return 'e', and the buffer will be "st". You can keep on reading until **Serial.available()** equals zero. (If you want to know the first byte without deleting it while doing so, you can use the **Serial.peek()** function.)

So the first `if(Serial.available > 0)` will check if there's any data in the buffer. If so, it will read the first byte, and store it in the variable `serialValue`. Then it will check if the value it just read is '0'. Note the single quotation marks, this indicates character zero (ASCII: '0' = 48), and not 0 as a value, since the serial monitor sends it as text. If the value is '0', it will turn off both LEDs. If the value is not '0', code in the **else** section will execute: so it will now check if the value is '1', if so, it turns on the first LED and turns off the second. If it's not '1' either, it will check if it's '2', if so, it turns on the second LED, and turns off the first one. If it's not '2', it will check if it is '3', if so, it turns on both LEDs, otherwise, it executes the code in the last else section, and prints what values you should enter.

You can check the flowchart in the image if the explanation wasn't clear enough.

Note that a double equality sign is used to check if two values are the same. If you would use a single equals sign, `( if(serialValue = '0') )` it won't check anything, it will just assign a value of '0' to the variable `serialValue`. This is a very common mistake.

Other operators to test values are `<` (less than) `>` (greater than) `<=` (less than or equal to) `>=` (greater than or equal to) `!=` (not equal to).

Inside of your if-statement, you can also use **logical operators (Boolean operators)**, to check multiple conditions: `&&` (and), `||` (or)

Some examples:

```
5 > 3 ? true  
5 < 3 ? false  
3 > 3 ? false  
3 >= 3 ? true  
5 != 3 ? true  
3 == 3 ? true
```

<http://www.instructables.com/id/A-Beginners-Guide-to-Arduino/>

to check if a value x is between 1 and 100:

`(1 <= x) & (x <= 100)`

another way of writing this (not recommended, just as an example)

`!( (x<1) || (x>100) )`

You can try to figure out how it works yourself, and then check with the truth tables in the image above.

Note: Just like in mathematics, brackets are used to indicate the order of operations, for example, in the last expression, 'x<1' will be tested first, then 'x>100', then '||', and finally '!'.

Arduino reference: While

Arduino reference: If (and comparison operators)

Arduino reference: Else

Arduino reference: Boolean operators

## Summary

- Most Arduinos have a second chip for USB communication. This chip communicates with the main microcontroller using a serial connection.
- Serial means that one bit is sent after the other, one at a time. There's a transmit line and a receive line (TX and RX respectively).
- You can use the Serial Monitor (CTRL+SHIFT+M) and the Serial Plotter (CTRL+SHIFT+L) to show the data the Arduino is sending, and to send data to the Arduino.
- `Serial.begin(baud)`; starts the serial communication with the computer. The default baud rate is 9600.
- `Serial.end()`; ends the serial communication.
- `Serial.print(text)`; prints text to the computer, it can be read in the serial monitor/plotter. Note that numbers are converted to ASCII: e.g. `Serial.print(0)`; will send a serial value of 48 (ASCII code for the zero character).
- `Serial.println(text)`; does the same as print, but will add a new line after the message.
- '\n' is a newline character, '\r' is a carriage return, and '\t' is the tab character (for indentations)
- `Serial.write(byte)`; sends a raw byte over serial. For example, `Serial.write(48)`; will print a 0-character in the serial monitor.
- `while(condition) { ... }` is called the while-loop. Code between the curly brackets will be executed and repeated as long as the condition between normal brackets is true.
- Serial returns true when the serial communication is active (when you open the serial monitor, for example).
- `while(!Serial)`; will repeat 'nothing' (read 'wait') as long as the serial communication isn't active.
- `if(condition) { if-code } else { else-code }` will execute the if-code if the condition is true, and execute the else-code if the condition is false.
- Serial data received by the Arduino is stored in a buffer, it stays there until you read it or until the buffer overflows.
- `Serial.available()`; returns the number of bytes available in the buffer.
- `Serial.read()`; will return the first byte in the buffer, and delete it afterwards.
- `Serial.peek()`; will return the first byte in the buffer, without deleting it.
- In your conditions, you can use these test operators: == (equal to), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), != (not equal to).
- And you can also use the logical && (and) and || (or) operators.

## Extra: Switch

In the last example, we used a lot of if ... else statements. Although this is the fastest way to do it, it's not easy to read. If you want to compare a variable to some given values, you can use a **switch**.

Open the example serialLEDsSwitch.

As you can see, the switch starts with the 'switch' keyword, followed by the variable you want to check between brackets. Between the curved brackets, our cases are defined. They use the syntax 'case value:', followed by the code you want to execute if the given variable equals the value in this 'case'. After the case-specific code, the 'break;' keyword is used to close the 'case'.

Note that the value you enter can not be a variable. If you want to compare 2 variables, you'll have to use if-statements.

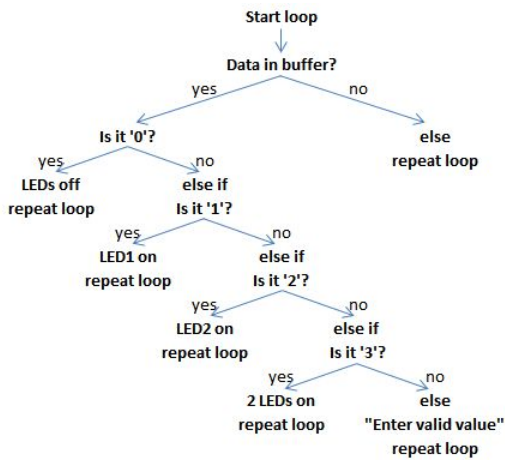
The code in the 'default:' case is executed if the variable doesn't match any of the values of other cases. This case is optional and can be omitted.

Arduino reference: switch / case

x	1 <= x	x <= 100	(1 <= x) && (x <= 100)
-999	0	1	0
0	0	1	0
1	1	1	1
50	1	1	1
100	1	1	1
101	1	0	0
999	1	0	0

x	x<1	x>100	(x<1)    (x>100)	!( (x<1)    (x>100) )
-999	1	0	1	0
0	1	0	1	0
1	0	0	0	1
50	0	0	0	1
100	0	0	0	1
101	0	1	1	0
999	0	1	1	0





x	1 <= x	x <= 100	(1 <= x) && (x <= 100)
-999	0	1	0
0	0	1	0
1	1	1	1
50	1	1	1
100	1	1	1
101	1	0	0
999	1	0	0

x	x<1	x>100	(x<1)    (x>100)	!(x<1)    (x>100)
-999	1	0	1	0
0	1	0	1	0
1	0	0	0	1
50	0	0	0	1
100	0	0	0	1
101	0	1	1	0
999	0	1	1	0

## Step 10: Analog inputs and outputs.

Some of the Arduino's I/O pins can also be used as digital inputs and outputs, to measure voltages (input) or dim LEDs (output) for example.

### Analog Inputs

Most Arduinos have between 6 and 12 analog input pins. They can measure voltages ranging from 0v to the input voltage (5v or 3v3)

An the board, these pins are labeled A0-A5, and in case of the Leonardo, some of the pins on the digital side can also be used as analog inputs. They are marked with a dot, and the name (A6-A11) is written on the back.

Note that reading an analog input is (relatively) slow, compared to reading digital pins. This is because the Arduino measures voltages using an internal reference voltage generator (DAC, Digital-to-Analog Converter), and then comparing the input voltage to the reference voltage, then changing the reference voltage, comparing again, changing the reference voltage, comparing ... until the two voltages are equal. <http://apcmag.com/arduino-analog-to-digital-conver...>

The resolution of the internal DAC is 10 bits. This means that the maximum number it can read is 10 bits long, or  $2^{10} = 1024$ , so a number from 0 (B000000000) to 1023 (B111111111). 1023 means the input is 5v (or 3v3) and 0 means 0v.

### Potentiometers

A potentiometer, variable resistor or pot for short, is just a knob with a wiper that slides over a round strip of resistive material. This way, it varies the resistance between the wiper and the endpoints.

We're going to hook up our potentiometers and faders as simple voltage dividers. You can read more on this [Wikipedia page](#) if you're not familiar with this principle.

If you take a look at the schematic, you can see 2 resistors. R1 is the resistor between the right pin of the potentiometer and the wiper (center pin), and R2 the resistor between the left pin and the wiper. Take a look at the formula as well. Since our potentiometer has a fixed value (50k?, for example), R1 + R2 will always be 50k?, while R2 can vary between 0? and 50k?. (If R2=50k then R1=0, and vice versa) Therefore, the fraction will always result in a number between 0 and 1. Multiply this ratio by 5V (Vsubin/sub), and you'll get a voltage between 0V and 5V on your output. This voltage can be read by the Arduino's ADC (analog to digital converter), and represents the position of the potentiometer (or fader). So basically, you connect the left pin to the ground, the right pin to the 5V pin on the Arduino, and the center pin to an analog input. (image 3)

Connect a potentiometer to pin A0, and open the example AnalogReadSerial (File>Examples>01.Basics).

The only new function is **analogRead(pin)**. It's quite self-explanatory, it just gives you the 10-bit value, representing the voltage on the given pin.

The program will just print them out over serial, so open the serial monitor (CTRL+SHIFT+M) or the serial plotter (CTRL+SHIFT+L) and twist the potentiometer. You should see values ranging from 0 to 1023.

Setting the delay to 10 or more, instead of 1ms may give you better results in the serial plotter.

You can map the values from the 0-1023 range to any other range, for example from 0 to 100. This can be done using the map function. Take a look at example AnalogReadSerialMap. The syntax of the map function is: **map(value, lowerLimitInput, upperLimitInput, lowerLimitOutput, upperLimitOutput)** value is just the value to map, in this case our sensor reading. The range of this input is 0 to 1023, so these are our input limits. We want the output to range from 0 to 100, so these are our output limits. This means that if the sensor reads 1023, the output value will be 100.

We could have achieved the same result by using  $\text{sensorValue} * 100 / 1023$ .

To convert it to a ratio, we have to use a data type other than int: **float** (floating point). If we divide our sensor value by 1023, we get a ratio between zero and one.

We can either change the data type of the sensorValue variable to float, or convert the value from an int to a float. The first method is used in AnalogReadSerialRatio-a, the latter one in AnalogReadSerialRatio-b.

If we don't convert our value to a float first, the result will be treated as an int as well. The values after the decimal point will just be ignored, for example,  $1/2 = 0$ , but  $1.0/2 = 0.5$ . Normal numbers like '1' will be treated as ints, unless you add a decimal point (1.0), then it will be treated as a float. If one of the factors in a calculation is a float, the result will also be a float.

To convert from an int to a float, you can use the **float(number)** function, or you could use the c++ notation for **typecasting** (converting from one data type to another) **(float) number**.

To convert from a float to an int, you can do this in the same way: **int(number)** or **(int) number**, but note that this will just truncate the number at the decimal point, it will just ignore the values that come after the decimal mark. For example,  $\text{int}(1.1) = 1$ , but  $\text{int}(1.9) = 1$  as well, even  $\text{int}(1.99999)$  will give 1. To round the number, use the **round(number)** function, For example,  $\text{round}(1.1) = 1$ ,  $\text{round}(1.5) = 2$ ,  $\text{round}(1.99) = 2$ .

<https://www.arduino.cc/en/Reference/Cast>

<https://www.arduino.cc/en/Reference/FloatCast>

### Measuring voltages

<http://www.instructables.com/id/A-Beginners-Guide-to-Arduino/>

Since an analog value of 1023 corresponds to 5v (or 3.3v for 3.3v microcontrollers), you can easily convert this to a voltage by using `analogRead(A0) * 5.0 / 1023`. This is used in the `AnalogReadSerialVoltage` example.

To read voltages higher than 5v (3.3v), you'll need 2 resistors, to create a voltage divider. We can transform the formula above to get  $V_{\text{subin/sub}}$  if we know  $V_{\text{subout/sub}}$ . (see image)

Connect one resistor (R2) from the ground to the analog input, then connect the ground of the voltage to measure to the Arduino's ground, and then connect another resistor (R1) from the voltage to measure to the analog pin. (see image) Use the formula above to calculate the appropriate resistor values. Any values between 10k and 100k should work fine. If the resistance is too low, it will draw too much current, and influence the reading a lot.

To measure 12v for example, I use  $R_1=47k$  and  $R_2=22k$ . This gives me a maximum voltage of 15.87v, if I'd go higher than this voltage, I'd break my Arduino.

Note: if R2 is - for whatever reason - disconnected or interrupted, the high voltage will be connected to the Arduino directly (through R1) and will probably damage or destroy it. So don't change the resistors if a high voltage is applied.

Open the example `AnalogReadSerialHighVoltage`.

At the top of the file, there are some **constant** declarations. Like a variable, constants can store values of all sorts, but unlike variables, constants can not be changed while the program is running. This means they are stored in the program storage space, instead of the dynamic memory (RAM), leaving more space for variables and arrays etc.

Change the values of these constants according to your setup. The ratio is calculated automatically, so you don't have to change it. You could use the color codes of your resistor to get a theoretical value, but you'll get a much better approximation if you measure the resistors using a multimeter. The same goes for the operating voltage.

## Analog outputs

The Arduino can't output analog voltages, it can only output either 5v or 0v. To output voltages in between, it uses a technique called **PWM (Pulse-Width Modulation)**. The Arduino creates a square wave, and then varies the on- and off-times of the wave. For example, 2ms on, 2ms off; or 1ms on 3ms off. This is referred to as the **duty cycle** of the square wave. 2ms on, 2ms off is a duty cycle of 50%; 1ms on 3ms off is a duty cycle of 25%. (see image)

You can now calculate the average voltage, which is the area under the curve divided by the amount of cycles. Let's take a look at the 50% duty cycle example: the area (evaluated over one cycle) is just a rectangle of 5 by 0.5, so the area is 2.5. Divided by the amount of cycles still gives 2.5, so the average voltage is 2.5v.

In other words, the average voltage can be written as the duty cycle multiplied by the supply voltage. For example, a duty cycle of 25% would be  $5v \cdot 0.25 = 1.25v$ . These average voltages are indicated in red in the images.

Your eyes are way too slow to see the underlying square wave, so this method is perfectly fine for dimming LEDs.

Open the example `analogPotDimmer`.

Connect an LED (+ resistor) to digital pin 5, and a potentiometer to analog pin A0. Turning the potentiometer dims the LED.

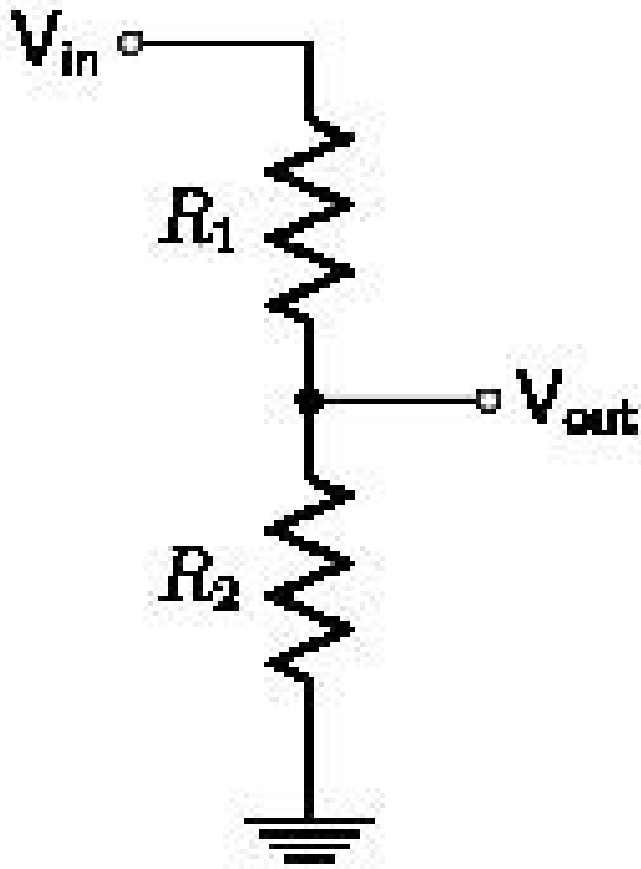
The only new function is **`analogWrite(pin, duty cycle)`**. It just sets the duty cycle of the pwm on a pin. Note that the duty cycle is a value from 0 to 255, this corresponds to 0%-100%

Another example is `analogWriteDimmer`. Connect a pushbutton to pin 2, and an LED to pin 5. When you press and hold the button, the LED gets brighter, when you release it, then press and hold it again, its brightness will decrease.

<https://www.arduino.cc/en/Reference/AnalogWrite>

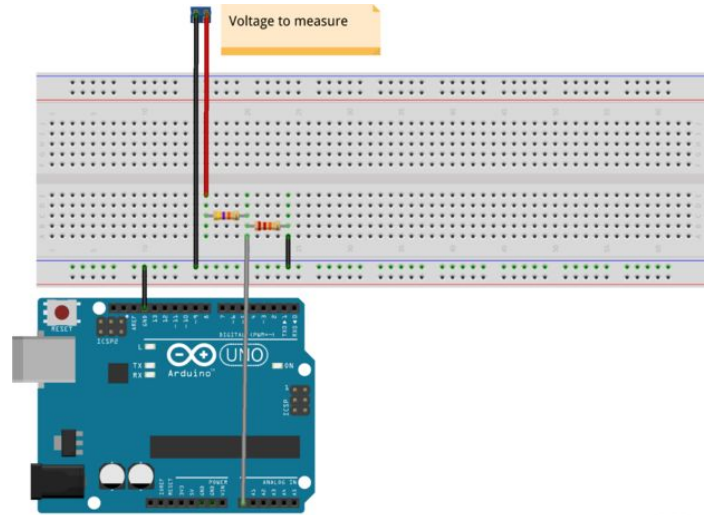
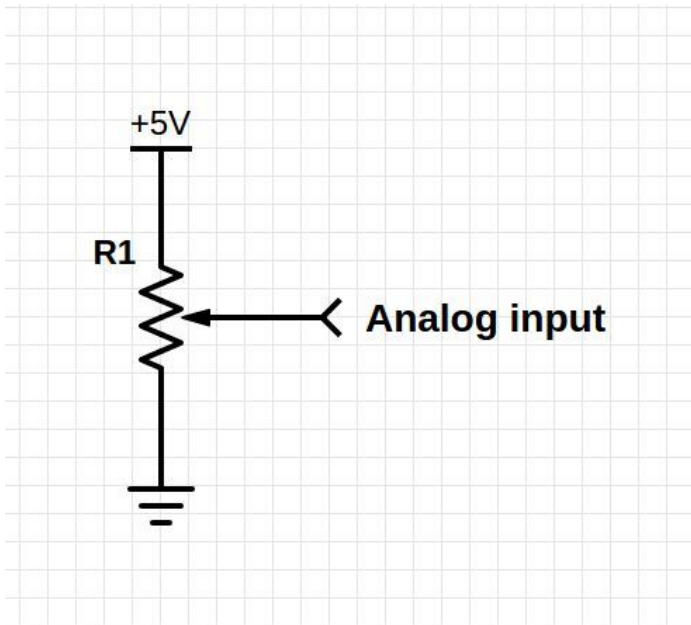
$$V_{out} = \frac{R_2}{R_1 + R_2} \cdot V_{in}$$

$$\Leftrightarrow V_{in} = \frac{R_1 + R_2}{R_2} \cdot V_{out}$$

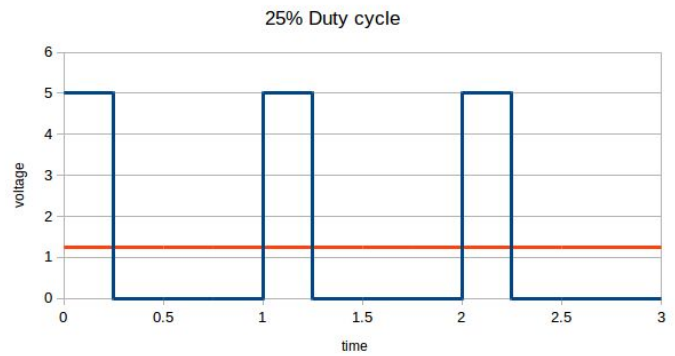
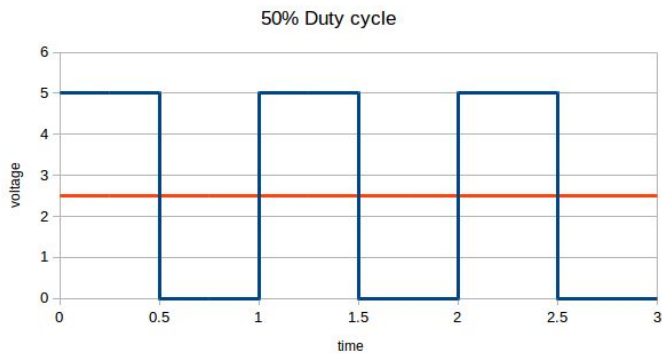


$$V_{out} = \frac{R_2}{R_1 + R_2} \cdot V_{in}$$

$$\Leftrightarrow V_{in} = \frac{R_1 + R_2}{R_2} \cdot V_{out}$$



fritzing



## Step 11: Driving more LEDs or other loads

Up until now, we only used components that draw very little current. But if we want to drive things that draw more than 20mA, the current supplied by the Arduino's output will not be sufficient. We'll need some sort of current amplifier, and that's when the **transistor** comes in.

When you want to drive things that run off a different voltage than the Arduino, you'll also need a transistor.

A small current from the base to the emitter results in a much higher current from the collector to the emitter. More information can be found in step 3.

### Measuring voltage and current

Before we can start to build our transistor circuit, we'll need to know the voltage and current draw of the load you want to use.

Connect your load (motor, fan, LEDs, bulb, heating element, solenoid ...) to the appropriate power supply, set your multimeter to DC volts and measure the voltage across the load, or directly from the load. Voltage is measured in parallel with the load. (see image)

Now disconnect the negative wire of the load from the power supply. Then plug the red wire of your multimeter into the amps connector, and set it to DC amps. Then connect the black wire of the multimeter to the ground of the power supply, and the red wire of the multimeter to the negative wire of the load. Current is always measured in series with the load. (see image)

Note: remember to plug your red wire back into the voltage connector of the multimeter, if you try to measure voltage with the current input, you will basically create a **short circuit**, and blow the multimeter's fuse or even **completely destroy it**. Some higher end multimeters will beep and show a warning when it's set to voltage while the current connector is used.

### Calculating the base resistor for the transistor

A transistor has a certain current gain, typically around 100. The symbol for **DC current gain** is a Greek letter beta ( $\beta$ ) or  $H_{FE}$ .

$$I_{\text{collector-emitter}} = I_{\text{base-emitter}} \cdot H_{FE}$$

Find the  $\beta$  value for your specific transistor in the data sheet. There should be a graph called "DC current gain". (see images) On the horizontal axis, you can find the collector current, this is the current drawn by the load. Note that most of the time, a logarithmic scale is used.

Write down the  $\beta$  value corresponding to your load current.

We need to know the base-emitter current, so divide the collector current by the current gain.

$$I_{\text{base-emitter}} = I_{\text{collector-emitter}} / H_{FE}$$

Now take the supply voltage of your Arduino, and subtract 0.7v. This is because the base-emitter silicon junction of the transistor has a voltage drop of 700mV.

Now use Ohm's law to calculate the resistance of the base resistor.

$$R_{\text{base}} = (V_{\text{Arduino}} - 0.7v) / I_{\text{base-emitter}}$$

Note: if the base current is greater than 20mA, you'll have to use a transistor with a higher  $\beta$  value or a Darlington transistor.

For example, I want to drive a motor at 200mA, 12v with a BD139 NPN transistor and a 5v Arduino:

As you can see in the second graph, at 200mA, the current gain is about 97.

$$I_{\text{base-emitter}} = 0.2A / 97 = 0.00206A \approx 21mA$$

$$R_{\text{base}} = (5v - 0.7v) / 21mA = 4.3v / 0.021A \approx 2048\Omega \approx 1.8k\Omega$$

In this case, it is better to round the resistance to a lower E12 value, to make sure the transistor is fully open.

When you use an inductive load, like a solenoid, relay, motor etc. you'll have to use a **flyback diode**. This is to prevent the transistor from being damaged by voltage spikes caused by the solenoid. (You get **self-inductance** inside of the coil when it is suddenly turned off.) (see image) You can use almost any rectifier diode, I used a 1N4007, for example.

Note: the Arduino's ground should **always** be connected to the ground of the transistor circuit.

Note: You can only use this method with low-voltage DC loads.

### Low-current MOSFETs

A normal (BJT, Bipolar Junction Transistor) as described above, is operated by the base-emitter current. MOSFETs (Metal Oxide Semiconductor Field Effect Transistors) are controlled by the gate voltage. (see step 3)

In the datasheet of your MOSFET, find the gate voltage to drain current graph. (see image, graph is for a BUZ11 MOSFET)

<http://www.instructables.com/id/A-Beginners-Guide-to-Arduino/>

As you can see, at 3.3v, the current is still pretty low, and useless in most cases. However, at 5v, the drain current might be enough for your specific application.

It is recommended to use pull-down (gate bleeder) resistor on the gate of the MOSFET to prevent an electric field from building up, and turning on the MOSFET, since it is very sensitive.

Just like with normal transistors, you should also use a flyback diode when switching inductive loads. (However, some MOSFETs have them built in)

(see images for schematics)

## High-current MOSFETs

To get higher currents, we have to get a higher electric field, so we need a higher gate voltage.

We could use transistor to do this, but it is easier to use an opto-coupler, or opto-isolator. This is basically an infrared LED and a phototransistor (light sensor) in one package. When the LED is turned on, the phototransistor conducts.

It looks just like an IC with only 4 (or 6) legs.

Using an opto-coupler also means that there is no electrical connection between the Arduino and the MOSFET, so if the higher-voltage circuit fails, it is almost impossible that it gets to the Arduino, and destroys it. This is a great advantage.

Take a look at the image for the schematic. You can use the formula for voltage dividers to calculate R1 and R2.

For example, if I want to drive a 12v 17A load with a BUZ11, I'll need a gate voltage of 6v (see graph).

6v is  $12v/2$ , so  $R1 = R2$ . They could be 47k?, for example.

In the formula,  $V_{in}$  is the supply voltage, and  $V_{out}$  is the gate voltage.

## Relays

To drive high-voltage or AC loads, you'll need a relay. See step 3 for more information.

The Arduino can't drive a relay directly, so you'll need a (small) transistor. A relay is an inductive load, so you'll need a flyback diode to protect your transistor. Use the method above to calculate the base resistor.

Take a look at the image above for the schematic.

**Warning: Wall power can kill you, if you're not careful enough. Never leave 115V or 230V connections exposed, and unplug your circuit when you're working on it.**

## Summary

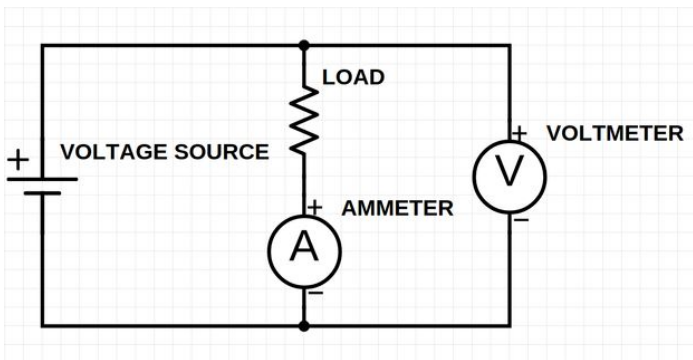
- If you want to drive things like motors or lights that draw more than 20mA or run at voltages other than 5v or 3.3v, use a transistor, MOSFET or relay.
- Always use a resistor on the base of a transistor to control the base current.
- Always use a pull-down resistor on the gate of a MOSFET.
- Always use a flyback diode when switching inductive loads.

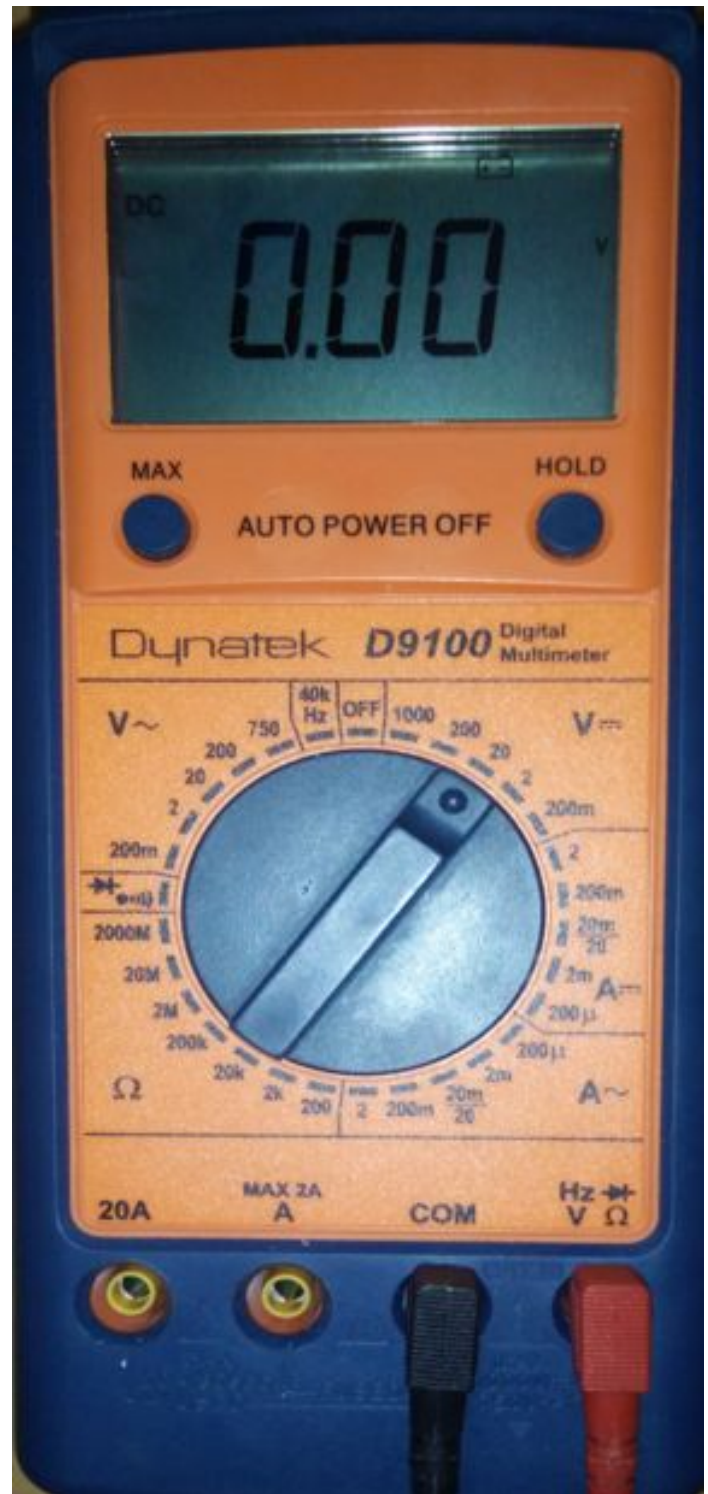
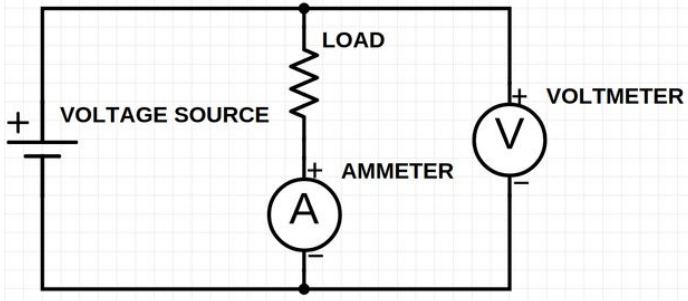
## Extra: PNP transistors

In the previous paragraphs, we only used NPN transistors, that take a positive signal to turn on. PNP transistors on the other hand turn on when a negative voltage is applied to the base, resulting in a "negative" base-emitter current. (negative, compared to the emitter, in a PNP transistor, the emitter emits "positive charges". It is still only the electrons that move, and a positive charge just means the absence of electrons.)

Take a look at the image above for the schematic. Just like with the NPN variant, the arrow at the emitter indicates the direction of the current.

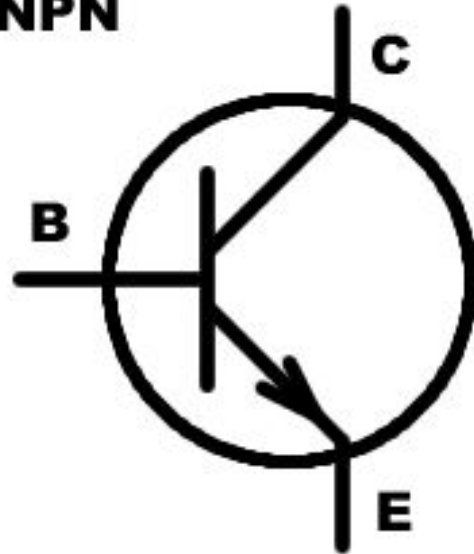
Note: the PNP transistor will conduct when the Arduino output pin is low.



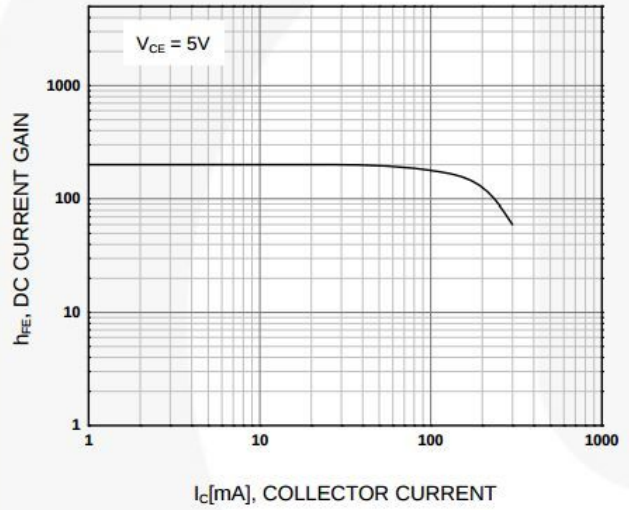
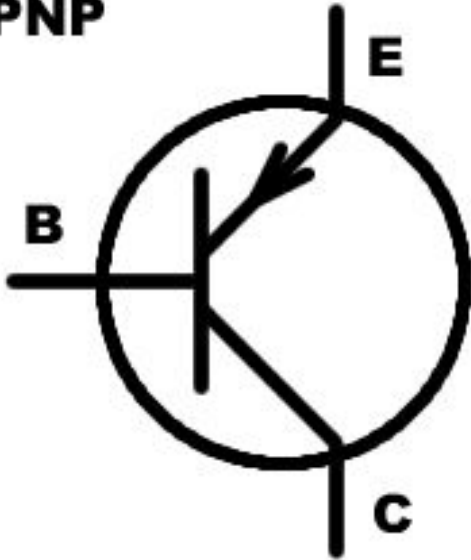




**NPN**



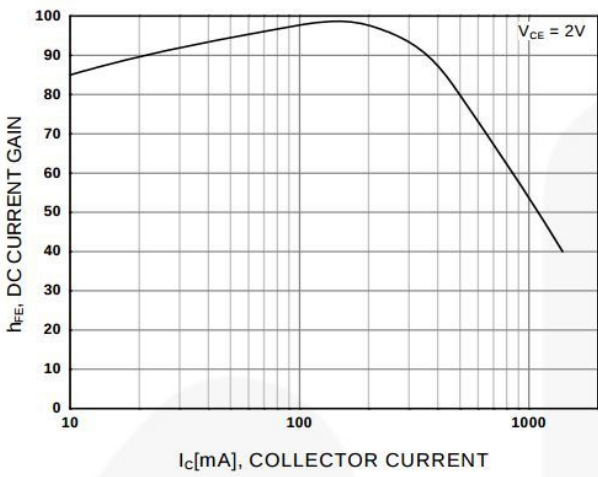
**PNP**



**Figure 3. DC Current Gain**

Image Notes

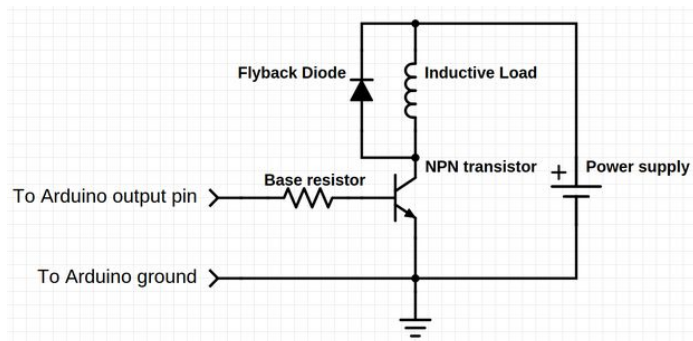
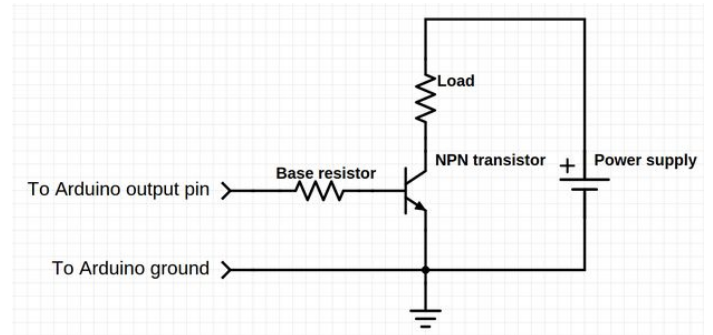
- 1. BC547



**Figure 1. DC current Gain**

Image Notes

- 1. BD139





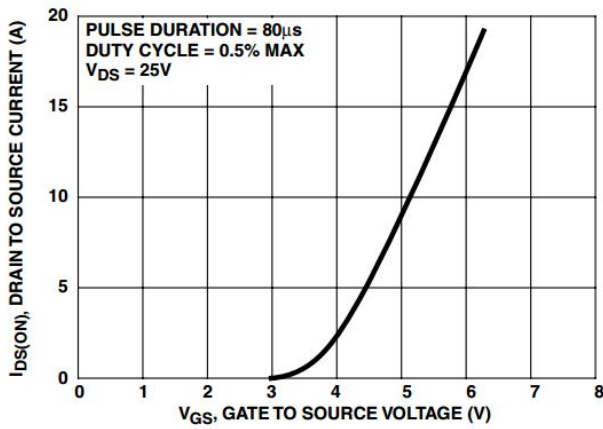
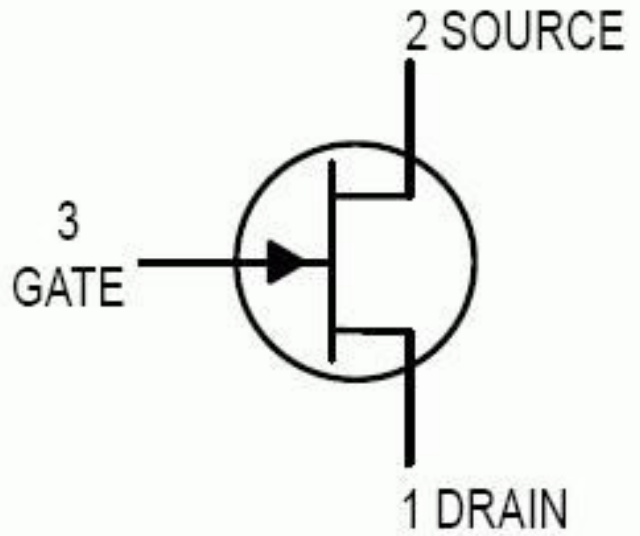
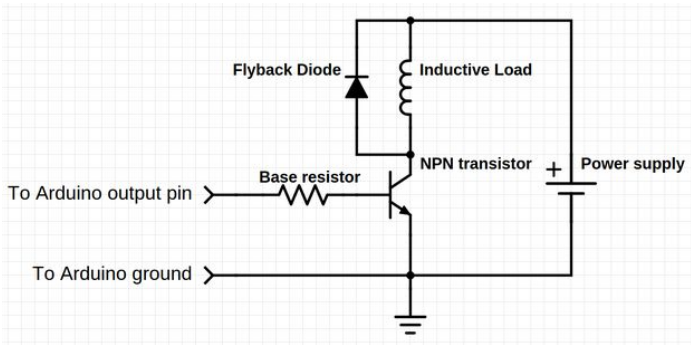
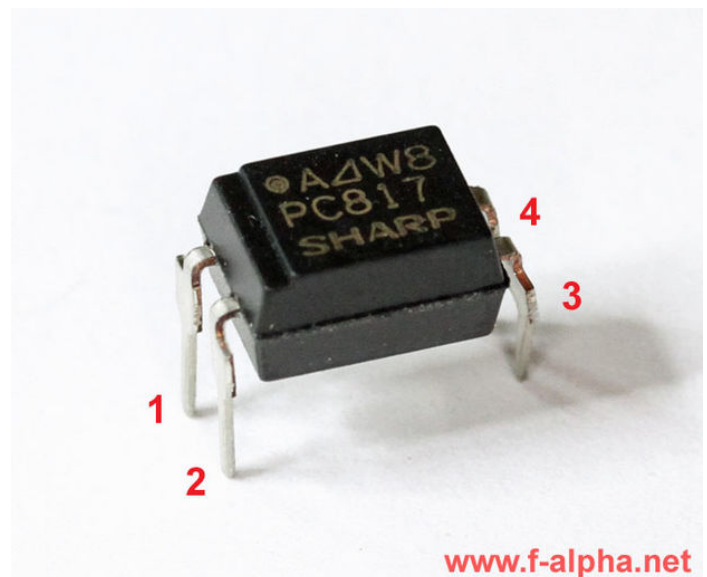
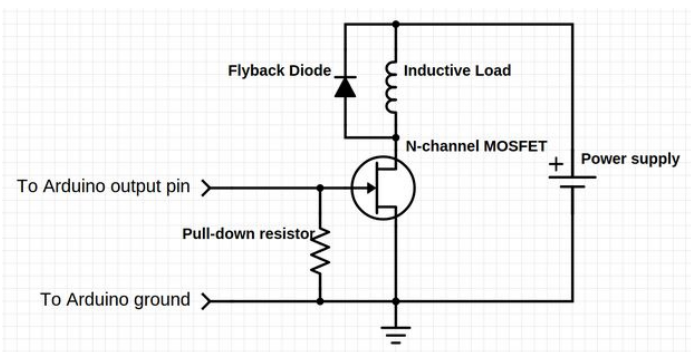
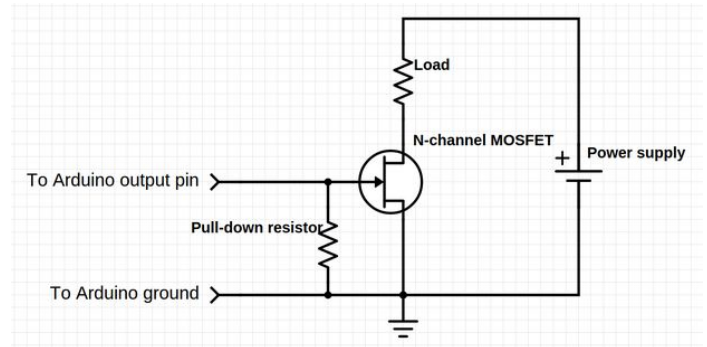
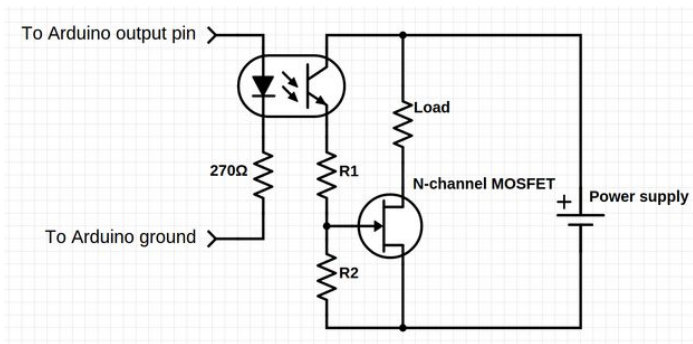


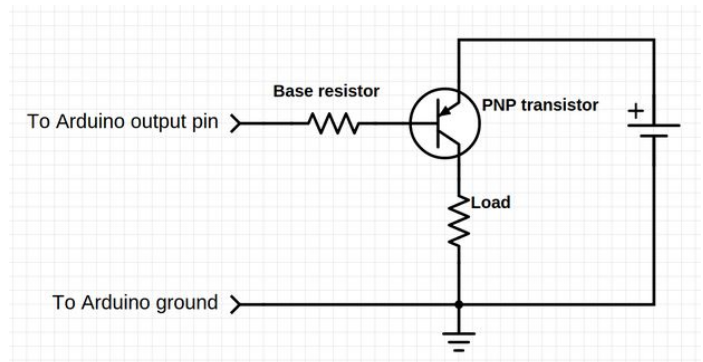
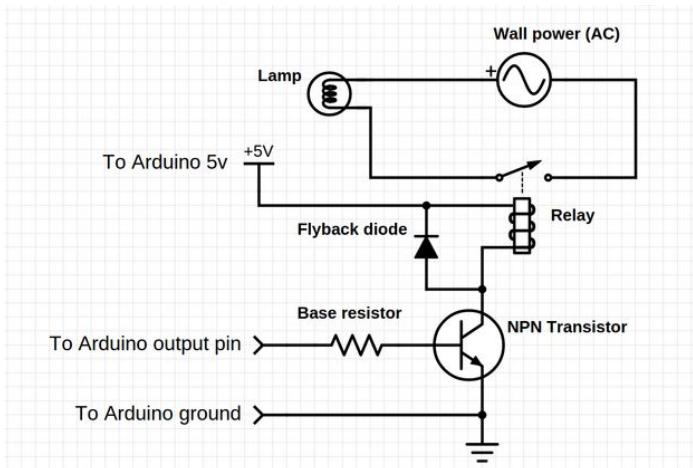
FIGURE 6. TRANSFER CHARACTERISTICS



[www.f-alpha.net](http://www.f-alpha.net)



$$V_{out} = \frac{R_2}{R_1 + R_2} \cdot V_{in}$$



## Step 12: Driving motors

### DC motors

With the techniques explained in the previous step, you can only spin the motor in one direction. To change the direction, you'll need an H-bridge (sometimes called a full bridge). It consists of 2 NPN and 2 PNP transistors, or 2 N-channel and 2 P-channel FETs.

Take a look at the image for a typical MOSFET H-bridge circuit.

Note: you can't just replace the MOSFETs in this schematic by transistors: Transistors need base **current**, FETs need gate **voltage**.

Let's find out how this circuit works:

We'll call the left input input A, and the right one input B.

When input A is low, transistor Q1 doesn't conduct. This means that the gates of Q2 and Q3 are high (they are connected to  $V_{SS}$  through R1). Q3 is an N-channel MOSFET, so it starts conducting (because it has a positive gate voltage). Q2 is a P-channel MOSFET, so it doesn't conduct (because it has no voltage difference between the gate and the source, both are connected to  $V_{SS}$ ).

The left terminal of the motor is now connected to ground, via Q3.

When input B is high, Q4 conducts. This means that the gates of Q5 and Q6 are low. (They are connected to ground through Q4, and R2 doesn't matter in this case). Q6 is an N-channel MOSFET, so it doesn't conduct (because it has no voltage difference between the gate and the source, both are connected to ground).

Q5 is a P-channel MOSFET, so it starts conducting (because it has a negative gate voltage, relative to the source).

The right terminal of the motor is now connected to  $V_{SS}$  through Q5.

In this case, the motor will spin counter-clockwise.

The second image is an equivalent circuit for this situation.

You can already guess that if input A is high, and B is low, the motor will spin clockwise. If A and B are both high or both low, the motor won't spin, because both terminals are connected to either  $V_{SS}$  or ground. (so no voltage difference)

Although you could build this circuit yourself, it is way easier to use an IC. For example, I got some L6202 chips out of an old printer. Another H-bridge chip is L298 or L293.

They connect directly to 2 digital pins on the Arduino, and most of them also have an 'enable' pin, which you can use with a third pwm pin, to control the motor's speed.

Adafruit has a really good tutorial on how to use them: <https://learn.adafruit.com/adafruit-arduino-lesson...>

### Stepper motors

Another commonly used motor is the stepper motor.

A normal DC motor has a spinning coil inside two permanent magnets. Stepper motors on the other hand, have a moving magnet (the rotor) and some stationary coils (the stator). By activating the coils in a specific sequence, the rotor will spin.

<http://www.instructables.com/id/A-Beginners-Guide-to-Arduino/>

The following videos will help you understand how it works:

For driving a **unipolar stepper**, you'll just need 4 NPN transistors, or 4 N-channel MOSFETs. (see image)

To find the pinout for your stepper, look up the datasheet, or use a multimeter, and measure the resistance between the wires. Some wire combinations will have a resistance that's 2 times higher than the others, these are the outer two wires in the schematic.

Connect the two center wires to the positive wire of the power supply, and the outer wires to the collectors of the drive transistors (see schematic). Calculate the appropriate base resistors, as explained in the previous step, and connect them to pins 8, 9, 10, 11 of the Arduino. Don't forget to connect the ground of the power supply to the ground of your Arduino.

For driving a **bipolar stepper**, you'll need 2 H-bridges. Connect the outputs of each H-bridge to one coil of the stepper. Connect the inputs of the H-bridges to pins 8, 9, 10, 11 of the Arduino. Don't forget to connect the ground of the power supply to the ground of your Arduino. Connect the enable line to  $V_{SS}$ .

Note: stepper motors draw a lot of current, so you cannot use the onboard power supply of the Arduino. You should also use high-power transistors or MOSFETs. Check the temperature of the transistors, and add a heat sink if necessary.

Now open the stepper\_oneRevolution example (File > Examples > Stepper) and change the number of steps per revolution to match your motor.

At the top of the file, there is a line

```
#include<Stepper.h>
```

This just adds the code of the Stepper.h file to the sketch, so you can use its functions. This is called the stepper **library**.

On line 24, we create an instance of the Stepper **class**. The instance is called 'myStepper'(this name is just arbitrary), and has 5 parameters: the number of steps in one revolution (360°), and the 4 pins that connect to the bases of the transistors.

A class has a set of **functions**, to run a function for a certain instance, a period (full stop) is used: **instance.function(arguments)**;

```
myStepper.setSpeed(60);
```

Remember Serial.print( ... )? This works in a similar way.

The **setSpeed(rpm)** function sets the speed, in rotations per minute (so it depends on the number of steps per revolution).

Another function of the Stepper class is **step(steps)**, it just turns the motor for a given number of steps. If this number is positive, it will move clockwise, if it's negative, it will move counterclockwise.

Upload the example to the Arduino, and the motor should turn 360° clockwise, then 360° counterclockwise, and so on. If it doesn't turn, but only vibrates, try swapping 2 output pins, until it works. You can swap the physical pins to the Arduino, or just change the pin order on line 24. (e.g. 9, 8, 10, 11 instead of 8, 9, 10, 11)

The usage of classes makes it very easy to create multiple Stepper instances. Take a look at example **\_2steppers\_oneRevolution**.

When a stepper is not moving, it will keep one coil on to block the motor, so it doesn't move. This draws a lot of current, however, and the stepper itself and the transistors may get very hot.

To stop this, you can just perform a digitalWrite(pin, LOW) on the 4 pins of the stepper.

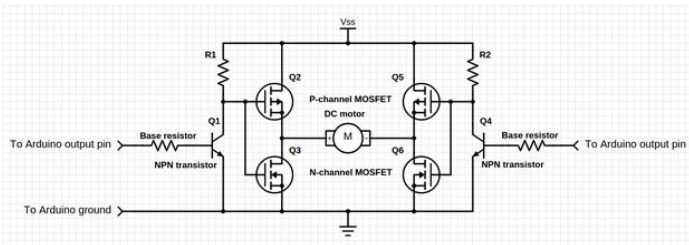
If you use a drive IC (like an H-bridge or a darlington array), you could also connect the enable pin to your Arduino. If you set it high, the motor will be activated, and when you set it low, there will be no voltage across the motor at all.

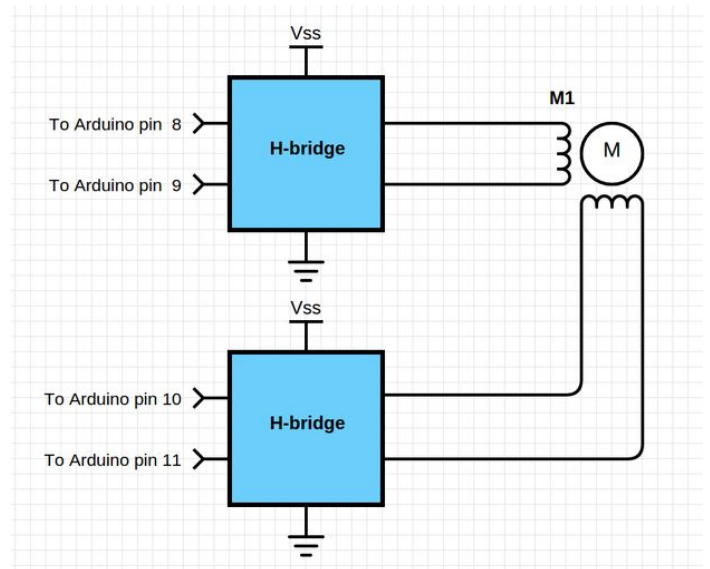
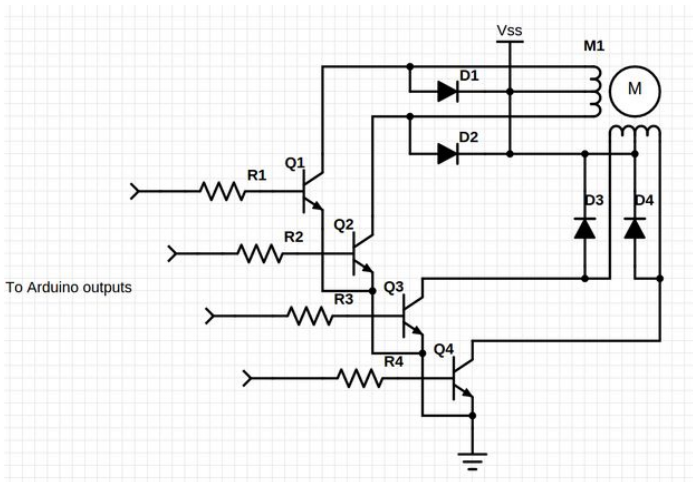
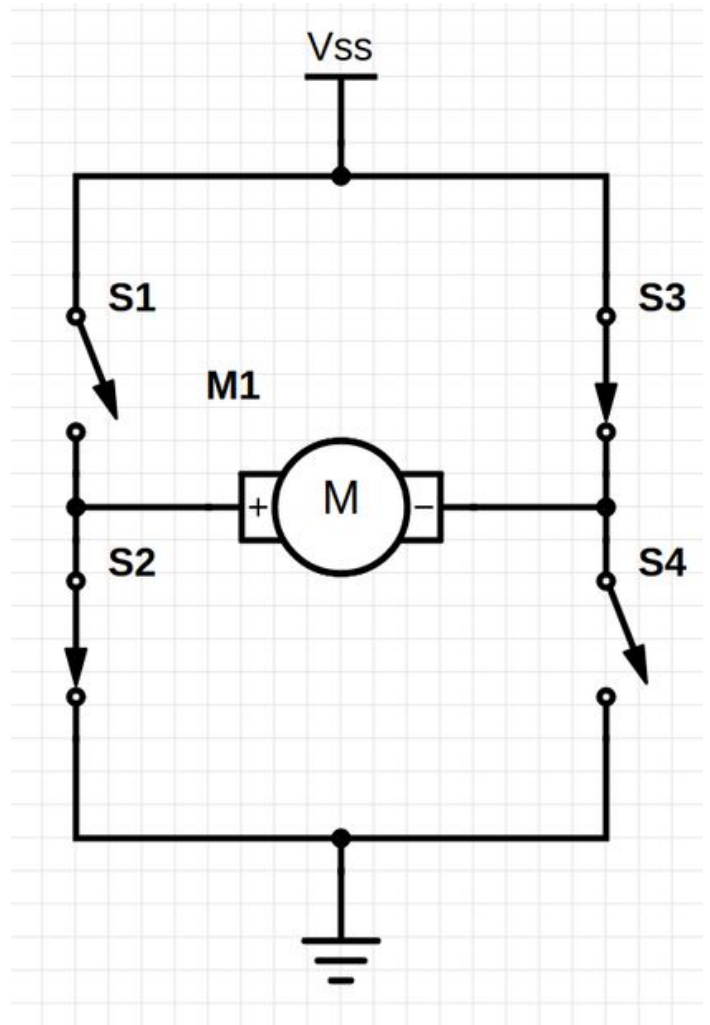
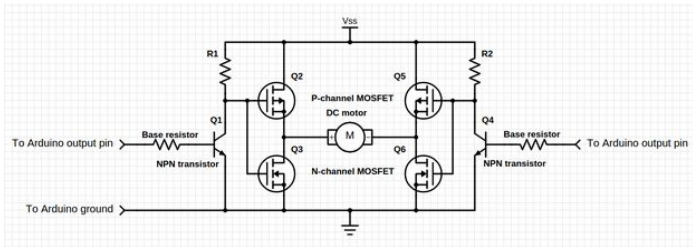
You could create your own little **function** to turn of the stepper, like I did in example Stepper\_stop

```
voidstepperOff(){  
digitalWrite(8,LOW);  
digitalWrite(9,LOW);  
digitalWrite(10,LOW);  
digitalWrite(11,LOW);  
}
```

You can make functions that do pretty much anything, for example if you have to execute a certain series of commands multiple times in your program. A good tutorial on classes, functions and libraries is the Library Tutorial on the Arduino site: <https://www.arduino.cc/en/Hacking/LibraryTutorial>

Arduino reference: Stepper library





### Step 13: Things not (yet) covered in this Instructable

There is of course no way to cover everything in only one Instructable, that's why I'll put some useful links to other tutorials on subjects I didn't include in this Instructable.

#### LCD displays

Arduino tutorial: Hello World

Arduino reference: Liquid Crystal

#### Servo motors

Arduino reference: Servo

Arduino tutorial: knob

#### Speakers and audio output

Arduino tutorial: Tone Melody

Google Code: Tone

#### Ultrasonic sensors

Arduino tutorial: Ping

#### The community

Arduino has an enormous online community, so Google will almost always give you some very helpful results. If you can't find what you're looking for, you can always ask for help on the forum. Just create a free account, it's really worth it!

### Step 14: Final thoughts

To conclude this Instructable, I would like to stress the importance of experimenting. You'll learn so much more if you try it yourself.

Find yourself a cool project, and go for it.

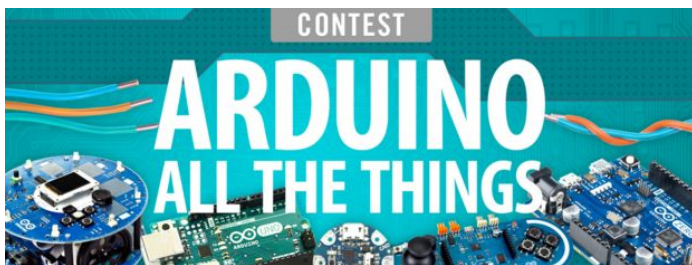
Try new things, stay curious!

If you liked this Instructable, I would like to ask you to consider voting for me in the **Arduino all the Things! Contest**.

If you have any requests, notes, things I missed, questions ... don't hesitate to use the comment section below ? !

Thank you for reading,

Tttapa, 25-01-2016



#### Related Instructables



**Simple Robotics Breadboard** by doctek



**Boxing The Teensy 3.1** by wpredator



**Persistence of Vision LED Display** by teamgoldteam



**Connecting NeuroSky's MindWave Mobile with PJRC's Teensy** by mvperr



**Refrigerator Lights** by JorgenVikingGod




**Custom MidiController** by JohnJ80


## Comments

2 comments [Add Comment](#)

---

 **jankar** says: Jan 26, 2016. 8:08 AM [REPLY](#)  
Havent received my Arduino yet but what appears to be a well thought out and very informative presentation is totally appreciated.  
Don't know how to vote but will when I figure it out.

---

 **ttapa** says: Jan 26, 2016. 8:24 AM [REPLY](#)  
Thanks a lot!  
(To vote, you can click the orange flag in the top right corner, or the trophy on the right in the header bar on mobile)

---