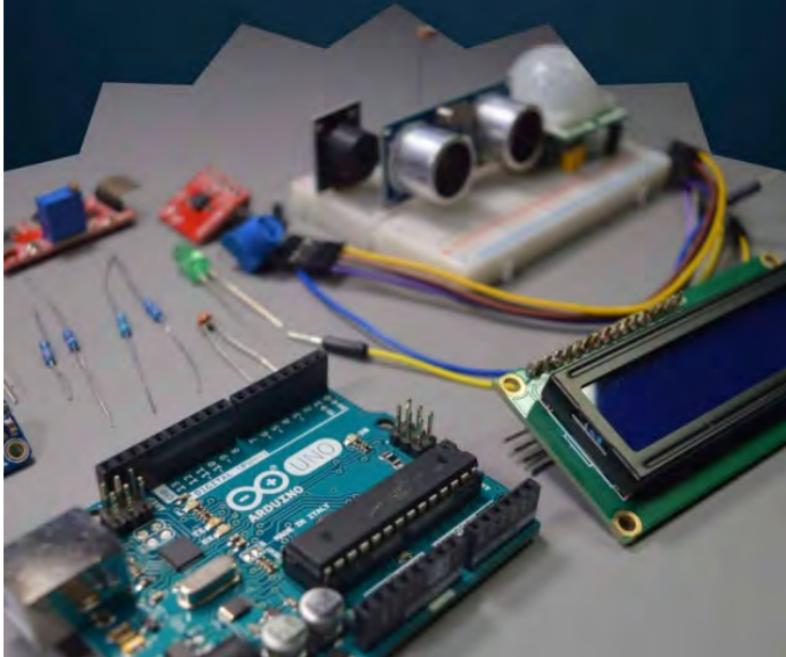




ARDUINO TIPS & TRICKS

GET THE MOST OUT OF YOUR ARDUINO WITH
ARTICLES FROM THE TECH EXPLORATIONS BLOG



Peter Dalmaris, PhD

Arduino Introduction and basic sensors

Get the most out of your
Arduino with articles
from the Tech
Explorations Blog

Welcome to this special collection of articles, meticulously curated from the Tech Explorations blog and guides. As a token of appreciation for joining our email list, we offer these documents for you to download at no cost. Our aim is to provide you with valuable insights and knowledge in a convenient format. You can read these PDFs on your device, or print.

Please note that these PDFs are derived from our blog posts and articles with limited editing. We prioritize updating content and ensuring all links are functional, striving to enhance quality continually. However, the editing level does not match the comprehensive standards applied to our Tech Explorations books and courses.

We regularly update these documents to include the latest content from our website, ensuring you have access to fresh and relevant information.

License statement for the PDF documents on this page

Permitted Use: This document is available for both educational and commercial purposes, subject to the terms and conditions outlined in this license statement.

Author and Ownership: The author of this work is Peter Dalmaris, and the owner of the Intellectual Property is Tech Explorations (<https://techexplorations.com>). All rights are reserved.

Credit Requirement: Any use of this document, whether in part or in full, for educational or commercial purposes, must include clear and visible credit to Peter Dalmaris as the author and Tech Explorations as the owner of the Intellectual Property. The credit must be displayed in any copies, distributions, or derivative works and must include a link to <https://techexplorations.com>.

Restrictions: This license does not grant permission to sell the document or any of its parts without explicit written consent from Peter Dalmaris and Tech Explorations. The document must not be modified, altered, or used in a way that suggests endorsement by the author or Tech Explorations without their explicit written consent.

Liability: The document is provided "as is," without warranty of any kind, express or implied. In no event shall the author or Tech Explorations be liable for any claim, damages, or other liability arising from the use of the document.

By using this document, you agree to abide by the terms of this license. Failure to comply with these terms may result in legal action and termination of the license granted herein.

#

1. What is the Arduino?
2. Common Arduino boards, problems and opportunities
3. Types of hardware that you can connect to an Arduino board
4. The Arduino programming environment
5. Arduino libraries and how to install them
6. The basics of Arduino programming:
program structure, functions, variables,
operators
7. The basics of Arduino programming: Loops,
conditions, objects, inputs & outputs
8. Introduction to Arduino sensors
9. Blinking LED
10. Fading LED
11. Button
12. Potentiometer
13. Infrared line sensor
14. Light sensor (analogue)
15. Impact sensor
16. Acceleration sensor
17. Ultrasonic distance sensor
18. PIR sensor
19. BME280
20. Measuring temperature and humidity

Copyright © 2024 by Peter Dalmaris, PhD

*For more educational content, please go to
<https://techexplorations.com>*

Tech Explorations creates educational products for students and hobbyists of electronics who rather utilize their time making awesome gadgets instead of searching endlessly through blog posts and YouTube videos. We deliver high-quality instructional videos and books through our online learning platform.

Lesson 1: What is the Arduino?

Introduction to the Arduino guide series

What is the Arduino?

This is the first article of the “Getting Started” series, where you will learn about the Arduino history.

This is the first article of the “Getting Started” series, where you will learn about the Arduino history. It’s even become a movie (well, ok, a [documentary](#))!

Let's begin...

I'm Peter, Chief Tech Explorer at Tech Explorations. I hope that this course will benefit you in two ways:

First, help you get started on your Arduino adventures, and over time assist you in your study of electronics.

Second, to give you a taste of my content and teaching style, with minimal effort required on your part.

And today, you will start with the short history and context of the technology education phenomenon, the Arduino.

But first, I want to dispel a myth. The myth is that, to learn electronics and to become a Maker, you need to be a naturally-born tech wizard.

This is important because a common complaint I hear from my students is that “I am not good with technology” or “my mind can't think like a computer.”

Unless you can deal with inner inhibitions such as these, it will be very hard, almost impossible, to become good at anything

that matters. There will always be an excuse.

Lack of learning inhibitions is one significant reason why children can learn so fast. There are other reasons, of course, but children, in general, haven't had time to develop defenses against learning. Everything is new, and as long as it doesn't look scary, as long as it's inherently interesting, children will go for it and learn.

What I have seen in my work with thousands of students, in my University career and at Tech Explorations, is that a learner's learning capacity is dampened primarily by negative prior experiences (like a bad experience in a classroom) or cultural accepted (but unexamined) norms (like "girls are not as good as boys in robotics").

How can you deal with something like this? How can you start to remove such inhibitions?

In my experience, you can do this in three steps:

1. Accept that these inhibitors exist (they do).
2. Accept that your capacity to learn is intact (it is).
3. Find convincing proof that counters each inhibitor (it exists).

I have been fortunate enough to have known some amazing people over the years:

A retired 65yo ex-police officer who had never programmed in his life until (in retirement), he decided to challenge himself with an Arduino. Now, he makes his own 3D printer and other gadgets.

A brilliant engineer with a severe mobility disability, who made her life's mission to create robotic technologies to assist people with mobility difficulties.

A pastor-teacher who reinvented himself as a STEM mentor powered by the Arduino because he believed that quality technology education allows his students to lead a fulfilling life.

These examples, and many more, show that learning is possible, as long as it is desired, and the right conditions exist. Each of the people in the examples had “valid” reasons to not learn. The ex-police officer could have considered his lack of prior programming experience and age to choose golf instead of electronics. The engineer with the mobility disability could have chosen a life dependent on support services instead of pursuing her studies despite her body’s deteriorating condition. And the pastor-teacher could have decided that STEM education is reserved for specialists.

At Tech Explorations, our mission is to help you create the conditions that can help you learn how to use the Arduino, regardless of where you are starting. Our courses provide a structured and distraction-free environment that can help you to focus on the task.

What you have to do is to make the decision that you want to learn, accept that you can learn, and create an environment that is right for you to do so. Our courses may (but don’t have to) be part of your learning environment.

Ok, now that we have the mindset sorted, let’s get into the Arduino.

What is the Arduino?

Let’s start at the very beginning. What is the Arduino, and where did it begin?

The Arduino is not a single thing. It is a prototyping platform. The platform is a collection of hardware, software, workflows (ways of doing things), and support networks (places where you can find help and information) designed to help people

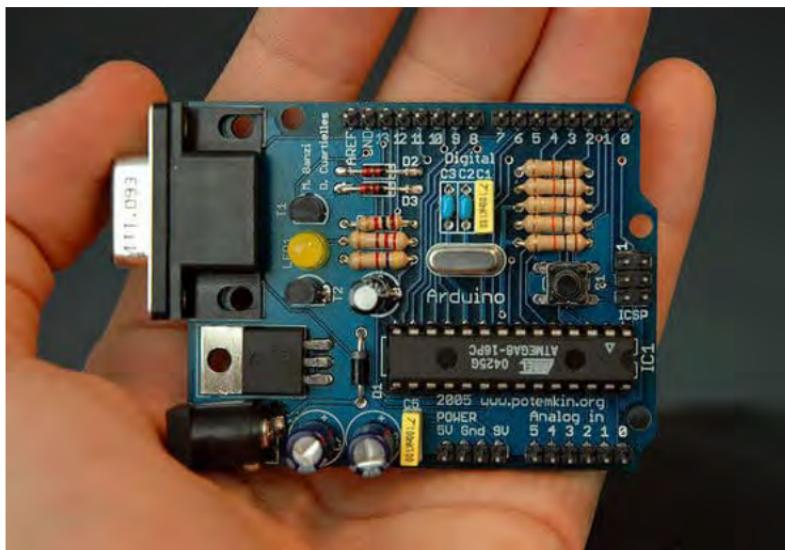
create prototypes (and often, finished products) very quickly.

All of these components are [open source](#), meaning that their designs and source code is available for anyone to copy and use.

At the center of the Arduino platform is a microcontroller chip.

A microcontroller is like a processor in your computer, except that it is very cheap, slower, and it has many connectors for peripherals like sensors and switches.

As a result, microcontrollers are great for sensing and controlling applications, and you find them everywhere: in your toaster, fridge, alarm system, in your car, printer, and paper shredder.



An early Arduino. It uses the RS232 serial interface instead of USB, an ATMEGA8, and male pin headers instead of female. The large black chip at the bottom right of the board is the microcontroller.

The Arduino was created by educators and students at the

Interaction Design Institute Ivrea in Ivrea, Italy. Massimo Banzi, one of the founders, was one of the instructors at Ivrea. At the time, students were using expensive hardware to build their micro-controller based creations.

The Ivrea students and their instructors decided to build their microcontroller platform by using a popular “AVR” microcontroller from Atmega, and a light version of the Wiring development platform written by (then student) Hernando Barragan.

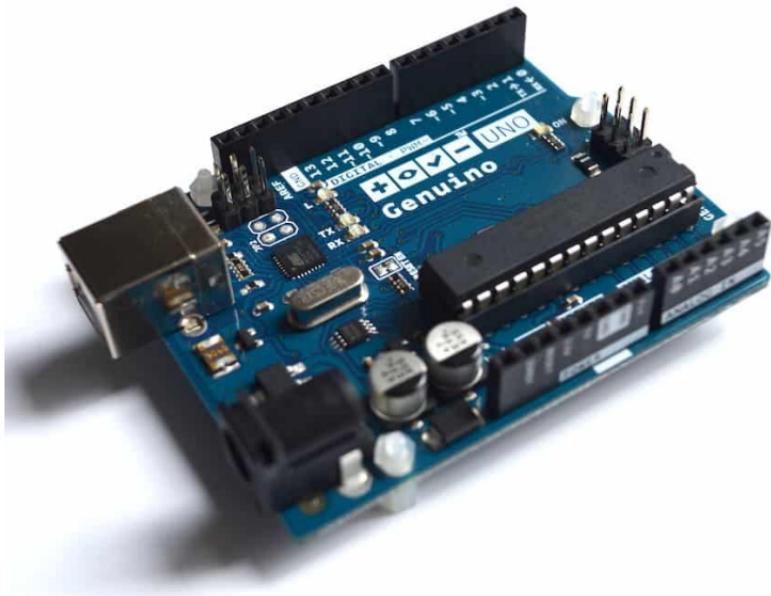
Wiring influenced the development of what we now call the “Arduino Language.”

The Arduino Language is, in fact, a simplified version of the C++ language. C++ is a general-purpose programming language used in the world’s most critical infrastructure, virtually all operating systems, desktop, and smartphone applications. Learning a bit of C++ is definitely not a waste of time.

The Arduino Integrated Development Environment (IDE) is also inherited from Wiring.

What the Arduino borrowed from Wiring made the platform easy to use so that people who are not engineers can build sophisticated microcontroller-based gadgets.

Tomorrow, we’ll be delving into useful resources for Arduino makers. These are places where you can look for inspiration and help as you get deeper into your Arduino prototyping adventures.



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

Lesson 2: Common Arduino boards, problems and opportunities

Introduction to the Arduino Guide series

Common Arduino boards, problems and opportunities

In this article, we take a look at some of the many kinds of Arduino boards that you can find in the market today.

In the [first lesson](#), you learned about the early days of the Arduino prototyping platform. I'd like to take this story a bit further and have a look at some of the many kinds of Arduino boards that you can find in the market today.

First: projects, problems, and opportunities

Before looking at the Arduino boards that make up the core of the Arduino ecosystem, I'd like to suggest that you look at the Arduino as a problem-solving tool, not only a learning or electronics tool.

The Arduino is truly a blank canvas on which you can design solutions to problems.

We often call these problems "projects." Some are big, some are small, but each problem is a project.

And each project (or problem) is an opportunity.

- An opportunity to learn something new
- An opportunity to re-examine our assumptions
- An opportunity to try a different way
- An opportunity to create something new

These opportunities come in many contexts. Education, technology, business, social, personal.

With the Arduino, you can capture these opportunities.

When you find a problem, in your mind, see “opportunity.”

Then, ask the question: “What kind of opportunity is this problem presenting me?”

Identify the opportunity. Consider its value. And if you decide that it is valuable, start the journey of capturing the hidden value.

This is the journey of solving a problem.

As you learn how to use the Arduino, you will encounter many problems. A sketch will not work as you expect it. A motor might spin in the opposite direction you thought it would. A sensor might give weird readings. It’s part of building stuff.

In reality, these are opportunities after opportunities, to learn things you didn’t know, to make things you haven’t made before, to think in a new way.

Often, you will become tired of this. You will be inclined to give up.

That’s when the word “opportunity” must light up your mind. The hardest things are more valuable, and capturing this value only comes at an expense: perseverance.

You can’t capture the value without experiencing the

hardships of the journey. You will have many such journeys with the Arduino, and each one will make you a little richer.

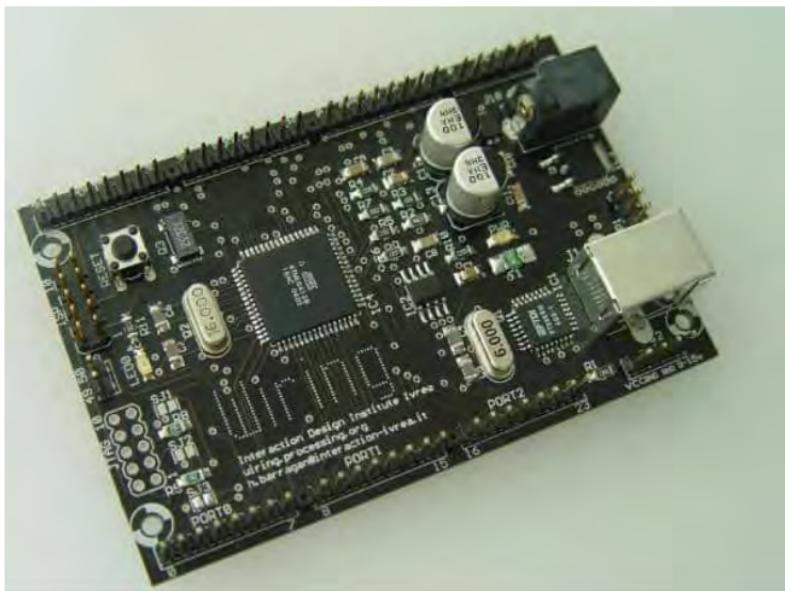
So, where were we?

Aha, the Arduino boards...

Wiring

Before Arduino became... Arduino, there was the Wiring Board.

The Wiring Board was designed by Hernando Barragán, then a student at Interaction Design Institute Ivrea (IDII). This board looked a bit like what a modern Arduino looks like. Massimo Banzi and Casey Reas were Hernando's thesis supervisors. You can learn more about this phase of the Arduino history in Hernando's [Untold Story article](#).



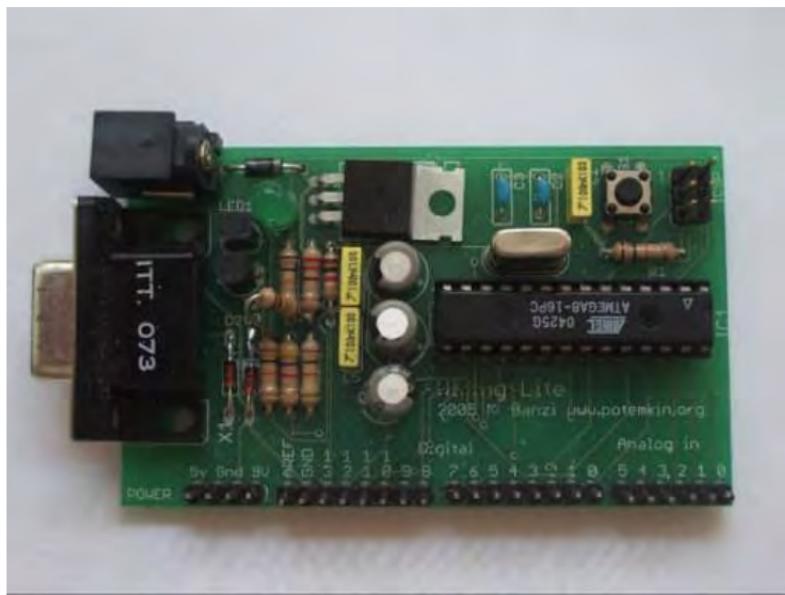
Hernando Barragán's Wiring Board, the Arduino's predecessor (original image available at <https://arduinohistory.github.io/>)

The Wiring Board contains the basic features so common in modern Arduino boards: a USB connection for uploading programs and for communications, an Atmega microcontroller, pins arranged along the edges, an onboard LED that you can control programmatically, and LED that light up when a program is transferred via the USB port.

This was in 2004.

Arduino

A year later, Massimo Banzi and David Mellis forked (a programming term that describes a copy of an original work) Wiring, and created the Arduino. They created the first Arduino prototype, pictured below:



Allegedly, the first Arduino prototype (original image available at <https://arduinohistory.github.io/>)

And shortly after, the second prototype:



The second Arduino prototype (original image available at <https://arduinohistory.github.io/>)

The second Arduino prototype is what we recognize today as the Arduino Uno, with several improvements as a faster microcontroller and better positioning of components, like the reset button.

Fast forward to 2019

Fast-forward to today, and in 2019 there are around 15 active and many more retired official Arduinos.

Because Arduino is open-source hardware, there are many more Arduino compatible boards, made by countless manufacturers. You can make your own, if you wish.

With so many Arduino boards out there, this is a topic that often confuses beginners.

In the remainder of this article, I'd like to help clear the

confusion.

As I mentioned already, there are dozens of boards designed by Arduino and various manufacturers. Some, but not all of these manufacturers, design their boards with care and attention to the official specifications to ensure that the boards they produce are fully compatible with the official Arduino boards.

The boards produced by these manufacturers are often labeled “100% compatible.” They are not official Arduinos (that means, they are not made by Arduino, the company), but they work exactly like an official Arduino.

There are also boards that are not fully Arduino compatible because their designer decided to make hardware changes that affect the way such board works. These boards are often called “clones,” and are usually much cheaper than an official Arduino. A common example is such boards that replace the USB components of the official board with a more affordable version. To use such Arduino on your computer requires you to install additional USB drivers, making this a wrong choice for beginners. The design of these boards is driven by price, so cheaper components are used all around; cheaper headers (where you connect the jumper wires), cheaper USB connectors, cheaper passive components (like capacitors and the voltage regulator). These Arduino clones cost a fraction of the cost of an official Arduino.

It is worth investing in official Arduino boards even if they are slightly more expensive because they will work better so that you will not have to spend hours figuring out problems with the board instead of building your gadget.

Arduino Uno

If you are a beginner in Arduino and electronics, I recommend getting the Arduino Uno R3; this is the classic Arduino board. It is hard to destroy by miswiring (I have tried!), has tons of high-

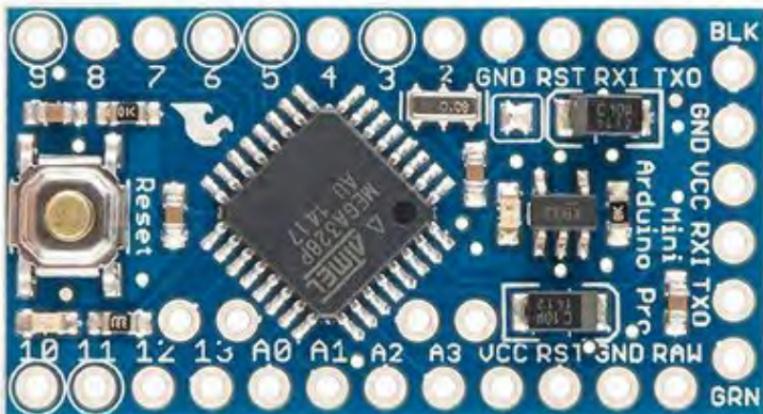
quality documentation, example sketches, and libraries while still surprisingly capable. It is relatively easy to expand as your projects grow.



This is the official Arduino Uno. The best board for the beginner and beyond (image taken from arduino.cc).

Arduino Pro Mini

If you are looking to build a project that requires a small size, you can go for one of the small footprint Arduinos, like the Pro Mini or the Micro (designed by SparkFun). These boards contain the bare-essential hardware. The Pro Mini is a personal favorite. You can find it on eBay for around \$5 (a bit more than the price of a single ATmega328P microcontroller), and it contains all of the functionality of the Uno except for the USB. It fits in the smallest project box, and I often attach it to custom-designed motherboards.



The tiny Arduino Pro Mini (image taken from arduino.cc)

The two example boards, the Uno and the Pro Mini, share the same basic architecture and are powered by the same microcontroller. This means that your circuits and sketches work the same way on these boards.

Arduino Mega

There are also Arduinos based on more capable microcontrollers, like the Mega, or even microprocessors running Linux, like the Arduino Yún rev 2.

The Arduino Mega 2560 is a super-sized Arduino Uno with a faster microcontroller (the ATmega2560) and many more input/output pins. This board is perfect for projects with a lot of buttons, motors, sensors, and, really, a lot of everything.



The Arduino Mega 2560 (image taken from arduino.cc)

Arduino Yún

The Arduino Yún rev 2, is essentially a computer combined with an Arduino. It runs a version of Linux, has built-in Wifi capability, and is made for Internet-of-Things applications.



The Arduino Yún, geared for the Internet-of-Things applications
(image taken from arduino.cc)

Arduino Gemma and LilyPad

Worth mentioning are also the various wearable Arduinos. If you are interested in making electronics that you can embed in clothing, then you can look at something like the super-tiny Arduinos Gemma or LilyPad (designed and sold by Adafruit). These are small, battery-ready, low power Arduinos, and circular so that they don't catch onto fabrics.



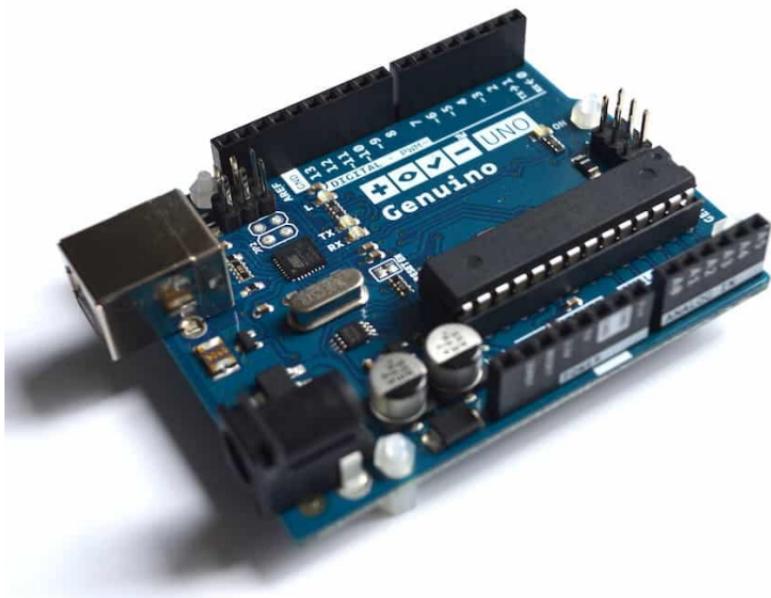
The super-tiny Arduino Gemma

So, which Arduino do I recommend for someone starting now?

Arduino is open-source, which means that its specifications are published so that anyone can create their own compatible and custom design.

It is easy to get lost in this variety of alternatives, but my advice is to start with the classic Arduino Uno.

In the next lesson, you will learn about the important hardware features of the Arduino Uno and the kinds of hardware that you can connect to it.



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

Lesson 3: Types of hardware that you can connect to an Arduino board

INTRODUCTION TO THE ARDUINO GUIDE SERIES

Types of hardware that you can connect to an Arduino board

In this lesson, we're going to dive into the types of hardware that you can connect to an Arduino board.

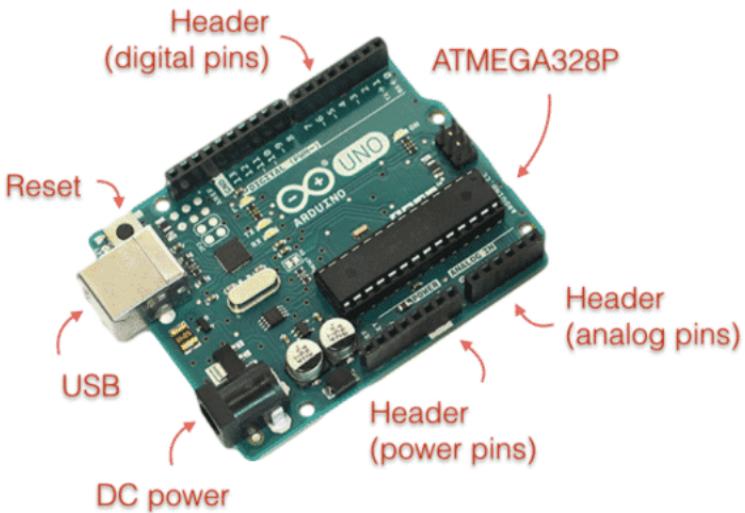
In the [previous lesson](#), you learned about the most common Arduino boards, and some of the history of the development of the first few boards.

In this lesson, we're going to dive into the types of hardware that you can connect to an Arduino board.

The Arduino can't do much on its own. Its purpose is to communicate with external hardware and to control.

There are many different kinds of hardware that you can connect to the Arduino. And there are a lot of them! In this section, I will discuss the kind of components that you can connect to an Arduino, and give some examples for each.

First, a quick look at the hardware included on the Arduino Uno board. The Arduino Uno has features that are shared with other Arduino boards. I'm marking the most important ones in this image:



The most important features of the Arduino Uno board

Here are the details:

- **USB:** The port used to transfer data and programs to the Arduino. It is also used to power the Arduino.
- **DC power:** If you don't connect the Arduino to the computer via the USB, you can power the Arduino by connecting a power supply or a battery pack to the DC power port.
- **Reset:** Press this button to make your program restart.
- **Headers.** There are four headers that expose pins. You can connect your peripherals to the Arduino using those pins.
- **ATMEGA328P:** This is the “brain” of the

Arduino Uno, the microcontroller. It sits on a socket, so if needed, you can swap it for a new one.

You will learn more details about the Arduino later. For now, you know enough to continue in this lesson and learn about the basic kinds of hardware that you can connect to the Arduino.

A simple mind experiment

As you read the rest of this lesson, do not be discouraged if the details seem hard to grasp. Unless you already have some knowledge of Ethernet controllers, transistors, and the like, you will need to take your time and learn all this, one step at a time.

Before you continue, do this mind experiment. Imagine yourself six months from now. You have completed this course, and you have completed the 'Arduino Step by Step Getting Started' course. Perhaps you are a quarter into the 'Arduino Step by Step Getting Serious' course.

You have already created Arduino gadgets that can communicate with the Internet; other gadgets that display sensor data on an LCD screen. One of your gadgets can turn the fan on when it's hot.

Your future self knows about shields, transistors, wifi modules, and LCD modules. You know how to write sketches to control all that, and to integrate them in a single working circuit. You feel confident that you can learn any new technology, and this confidence stems from your recent achievements.

Not only that, but your collection of boards and components has grown. Thanks to eBay, Aliexpress, Amazon, and many other international retailers and global trade, these components are really cheap to buy. You don't need to take

apart and reuse components constantly. Experiments never fail, and even if you burn out an LED, you can easily replace it and continue with the next experiment.

Now, come back to the present.

You are just starting out.

Don't lose sight of your future self that you want to be.

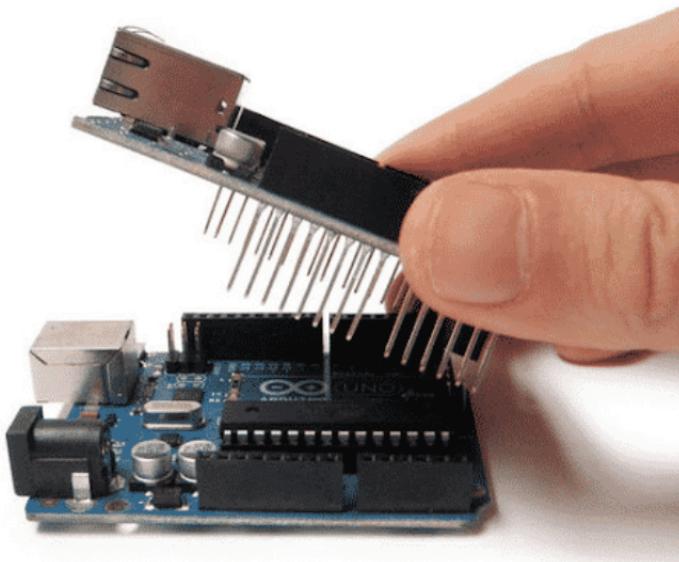
Commit to the journey to take you there.

Shields Up!

Shields

An Arduino shield is a printed circuit board with various components already installed on it, ready to perform a particular function. They hook onto an Arduino without any wiring or soldering. Just align the shield with the Arduino, and apply a bit of pressure to secure them.

Most shields are built to work with the Arduino Uno, and as a result, virtually all other full-sized Arduinos have an Uno-compatible header configuration.

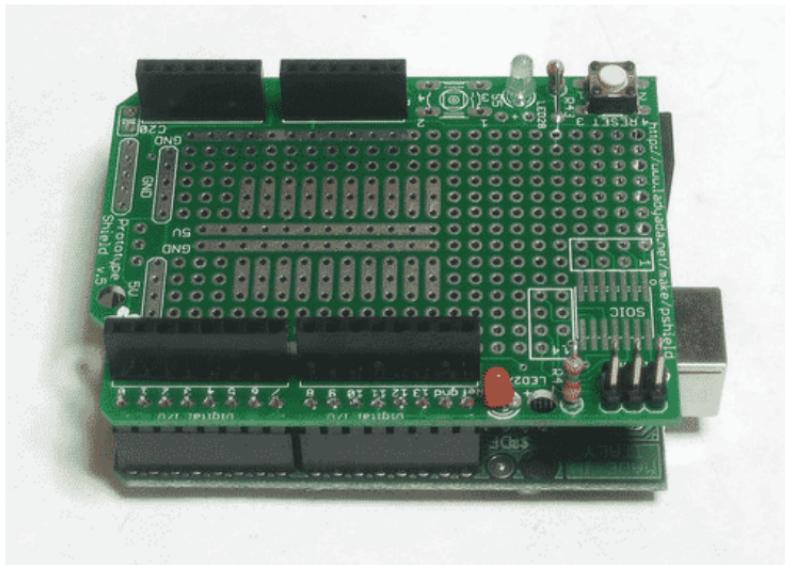


An example Arduino Uno shield. It fits perfectly in the Arduino Uno headers and adds capabilities without any jumper wires.

The Arduino Ethernet shield (top) is about to connect to an Arduino Uno (bottom). To make the connection, align the pins of the shield with the headers in the Uno and gently press down.

There are shields for almost anything: Ethernet and Wifi networking, Bluetooth, GSM cellular networking, motor control, RFID, audio, SD Card memory, GPS, data logging, sensors, color LCD screens, and more.

There are also shields for prototyping, with which you can make permanent any circuits you created on a breadboard and are too good to destroy.



A prototyping shield has enough space and pads where you can attach your own components.

A prototyping shield like this one from Adafruit makes it easy to preserve your best circuit designs.

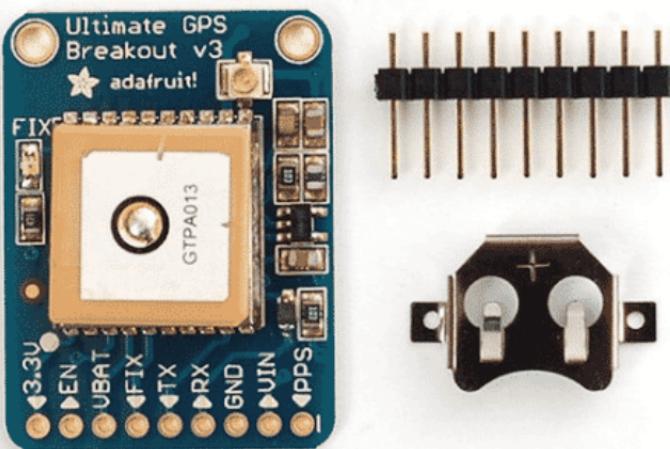
Shields are great for beginners because they require no tools to add components to an Arduino.

Breakouts

Breakouts are typically small circuit boards built around an integrated circuit that provides specific functionality. The board contains supporting circuitry, like a subsystem for supplying power, LEDs for indicating status, resistors and capacitors for regulating signals, and pads or pins for connecting the breakout to other components or an Arduino.

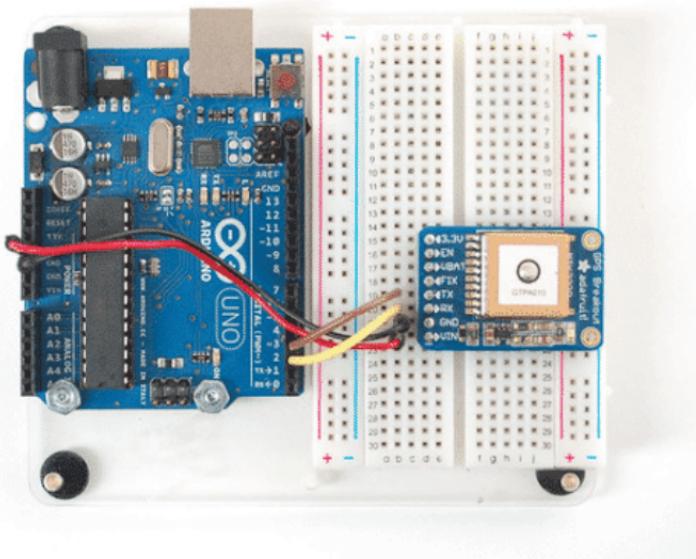
In many cases, the same functionality is offered in a shield or a breakout format. For example, you can get the same GPS as a breakout or as a shield. In such cases, the difference is size. The breakout is smaller; it can work with boards other than the

Arduino Uno or Arduinos with the Uno headers.



The Adafruit GPS Breakout. It comes with a header and a battery holder that you must solder on (image courtesy of Adafruit).

A breakout has to be wired to an Arduino using jumper wires and often a breadboard.



You must connect the breakout to the Arduino using wires and a breadboard (image courtesy of Adafruit).

Sometimes, apart from using jumper wires to connect the breakout to the Arduino, you may also need to do a bit of soldering, like I had to do for the GPS Breakout. Here's the quick version of how this soldering job went (you can see me soldering the Adafruit GPS breakout in the fast-motion video below).

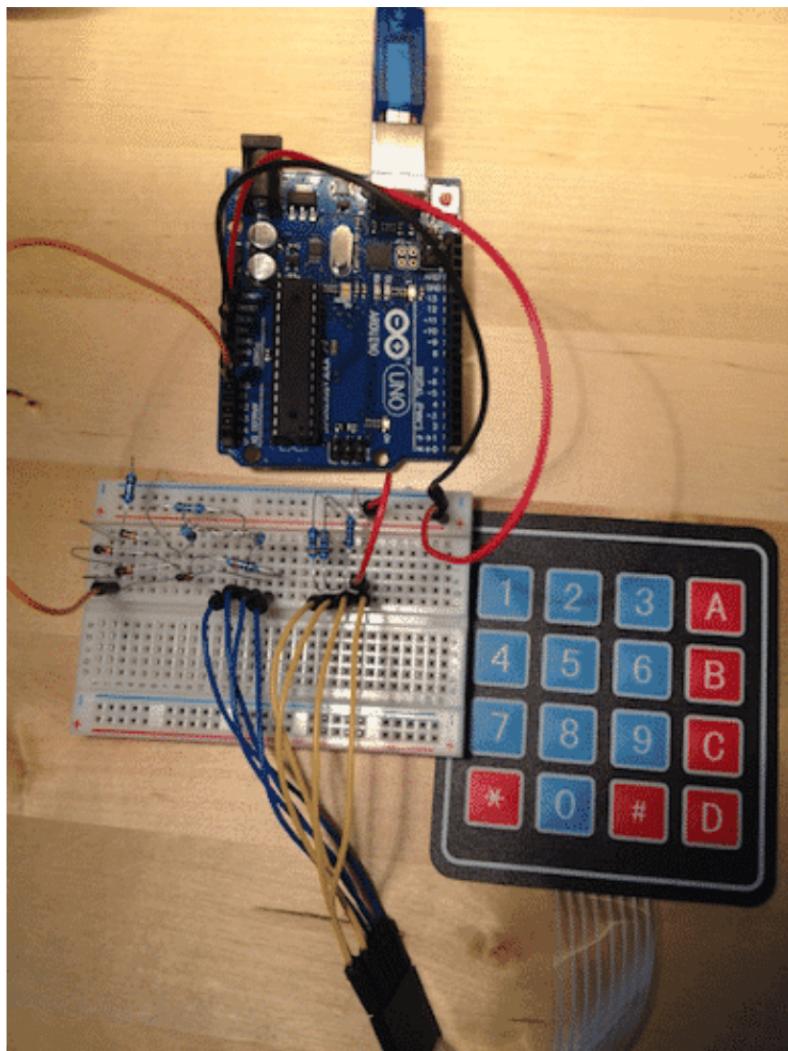
The beautiful thing about breakouts is that unlike shields, which only work with the Arduino, a breakout can be connected to anything, including the boards that you will design yourself down the track. Therefore, apart from being used for learning, breakouts can be embedded into a final product.

Components

While breakouts give you easy access to components by putting them on a printed circuit board with their supporting

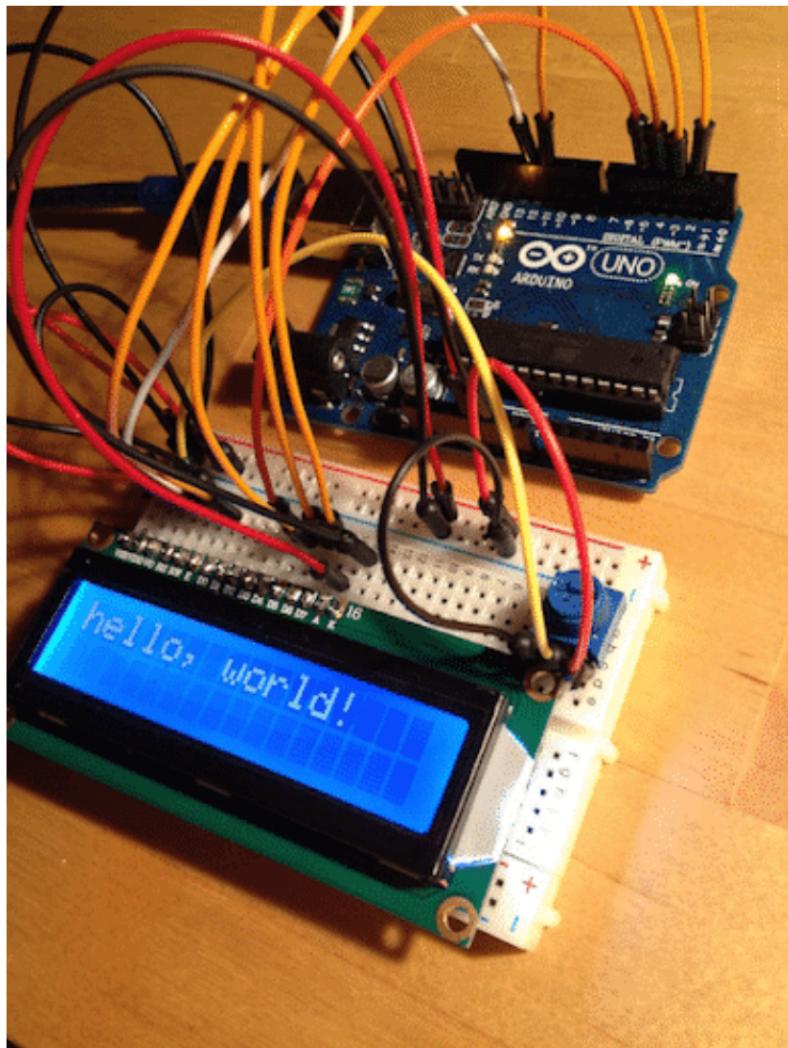
electronics, you will eventually need access to the individual component so that you can fully customize the way it works in your circuit.

For example, if you would like to have a keypad so that the user can type letters and numbers as input to a gadget you are making, you could use a soft membrane keypad. This keypad is available as a component. To use it correctly, you will need to add several wires and resistors.



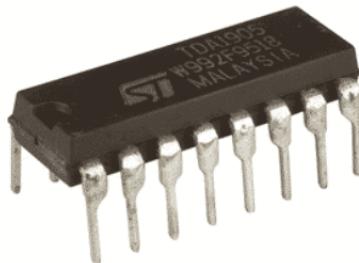
Using a 4x4 keypad requires external wires, diodes, and resistors; this is more work (compared to a shield), but often the flexibility you get in return is worth the effort.

Another example of an individual component is a character LCD screen. To make this one work properly, you have to provide a lot of wires and a potentiometer.



An LCD screen on a breadboard. A lot of wires are used to

connect it to an Arduino Uno, on a breadboard with a potentiometer.



A shift register IC makes it possible to control many digital components with a single pin of your Arduino.

As you become more skilled in Arduino prototyping, you will find yourself using increasingly more components like these. Almost any functionality you can imagine is available as a component. Sensors of all kinds, motion, user input, light, power, communications, storage, multiplexing and port multipliers, binary logic integrated circuits, amplifier circuits, even thumbprint scanners can be connected to an Arduino as components.

Discrete components

At the bottom of the scale, regarding size and complexity, we have a wide range of discrete components. Things like resistors, capacitors, transistors, LEDs, relays, coils, etc. fall into this category. They are the “brick and mortar” of electronics. Most of these discrete components are very simple but very important.

For example, a resistor limits the amount of current that can

flow through a wire. A capacitor can be used as a small store of energy or as a filter. A diode limits the flow of current to a single direction. An LED is a diode that emits light. A transistor can be used as a switch or an amplifier. A relay can be used to switch on and off large loads, like an electric motor. A coil can also be used as a filter or as part of a sensor, among other things. There are many more discrete components that the examples mentioned.



A resistor limits the amount of current that flows through a wire.



A capacitor stores energy, or works as a filter.



A diode limits current to flow towards one direction only.



An LED is a diode that emits light.



A transistor can be used as a switch or an amplifier.



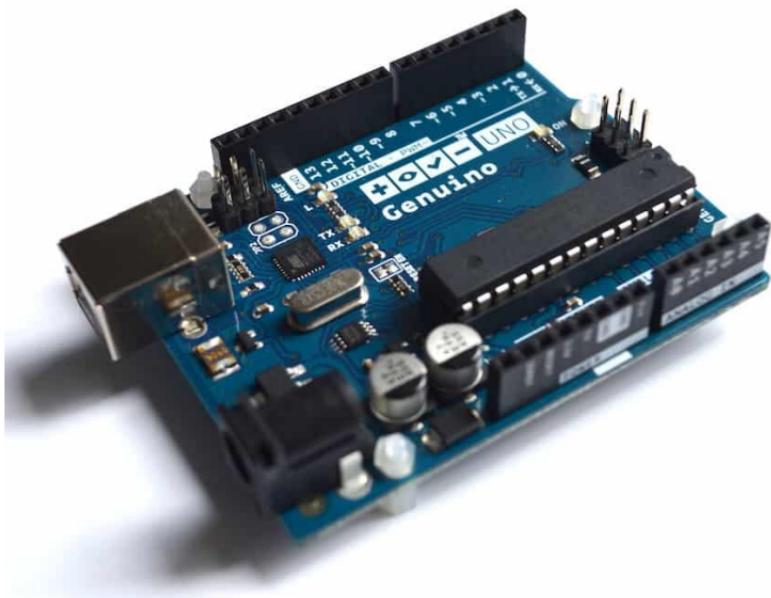
A relay is used to drive large loads from your Arduino.



A coil can be used as a filter.

As you are starting your electronics adventures, no matter which Arduino you choose, you will need to stock up on these components as you will need to use them in virtually everything you make. Luckily, they are very cheap, and it is worth buying them in bulk so that you always have some when you need them.

Now that you have a better understanding of the type of hardware that you can connect to an Arduino board, you are ready for the next installment of this short course. In the next lesson, you will learn how to install the Arduino programming environment on your computer and its basic features.



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

Lesson 4: The Arduino programming environment

Introduction to the Arduino guide series

The Arduino programming environment

In Lesson 4 of our introductory course on the Arduino, you will learn about what it is like to program the Arduino. You will install the Arduino IDE, and use it for the very first time to upload your first example sketch.

In Lesson 4 of our introductory course on the Arduino, you will learn about what it is like to program the Arduino. You will install the Arduino IDE, and use it for the very first time to upload your first example sketch.

You will need your Arduino Uno and a USB cable.

I recommend that you clear the next 15 minutes in your schedule so that you can concentrate on the activities of this lesson. Switch your mobile and computer to “do not disturb” mode.

You don’t need any prior knowledge or skill in electronics. You don’t need any prior knowledge or skill in programming.

By the time you finish this lesson, you will have a programming environment installed on your computer, and an Arduino with a blinking LED.

And that’s your first practical step in learning the Arduino.

Right now.

Let's do it!

The Arduino quick setup guide

To program the Arduino, you need to install the Arduino Integrated Development Environment (IDE) on your computer.

To install the IDE on your computer, you must first download it from arduino.cc. Then, you will follow the installation process that depends on the operating system that is running on your computer.

The only requirement is that you have a Java runtime environment already installed. This is usually not a problem on Windows and Mac computers.

Download the software

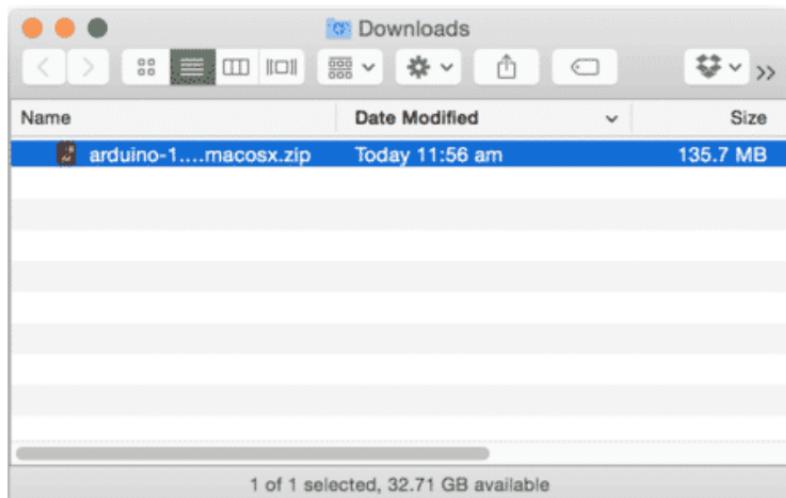
To download the IDE for your operating system, go to <https://www.arduino.cc/en/Main/Software>.

The screenshot shows the Arduino website's download section. At the top, there's a navigation bar with links for HOME, STORE, SOFTWARE, EDU, RESOURCES, COMMUNITY, and HELP. A magnifying glass icon for search and a 'SIGN IN' button are also present. Below the navigation, the text 'Download the Arduino IDE' is displayed. To the left is a large circular icon containing a minus sign and a plus sign. In the center, there's a section titled 'ARDUINO 1.8.8' with a brief description of the software. To the right, there are download links for various operating systems. An orange rectangle highlights the 'Windows Installer' link for Windows XP and up. Below this, there are links for Windows app, Mac OS X, Linux 32-bit, Linux 64-bit, and Linux ARM. At the bottom of this section, there are links for Release Notes, Source Code, and Checksums (SHA256). On the left, there's a 'HOURLY BUILDS' section with a 'LAST BUILD' link. On the right, there's a 'BETA BUILDS' section with a 'LAST BUILD' link. Both sections have download links for Windows, Mac OS X, Linux 32-bit, and Linux 64-bit.

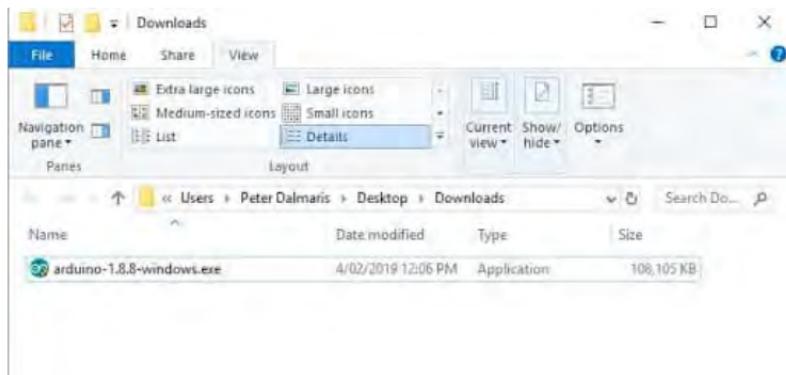
The Arduino IDE download page. Pick the installer to match your computer operating system (download links are in the orange rectangle).

In the download page, scroll down to find the latest available version, and click on the download file link that matches your operating system. For Windows, I prefer the ZIP file option.

Allow some time for the download to complete. Then, look for the installation file in your download folder, “~/Downloads” for the Mac and “Downloads” on Windows.



The IDE installer on the Mac

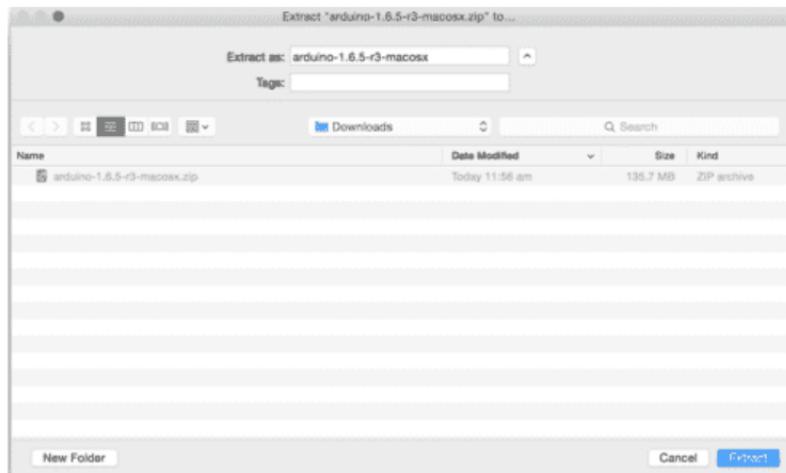


The IDE installer on Windows 10

In the remainder of this lesson, I show you the installation process for Mac OS and Windows 10.

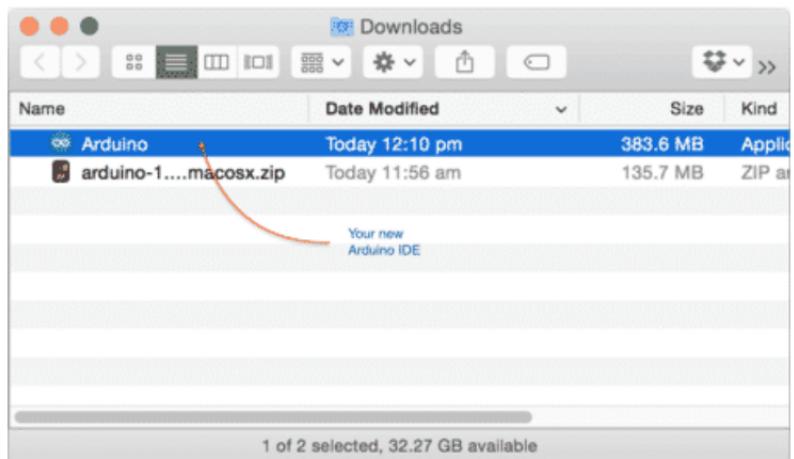
How to install the Arduino IDE on Mac OS X

To do the installation on the Mac, double-click on the installation archive file to have it extracted. The picture below may be different from what you will see, depending on which program you use for extracting ZIP files.

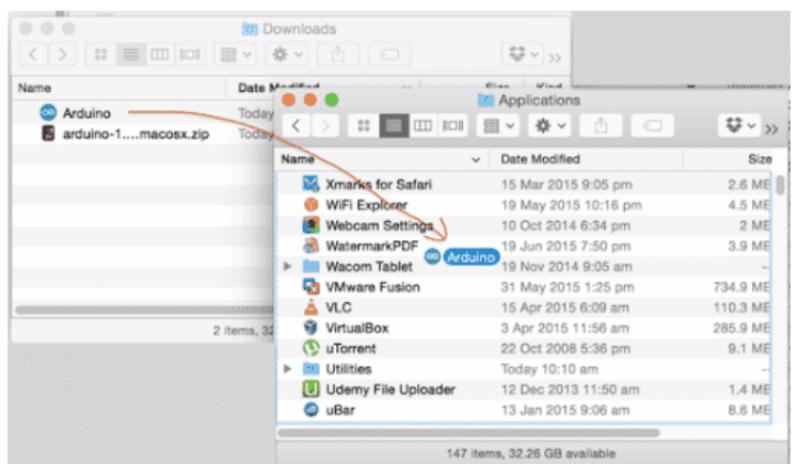


Extracting the IDE installer on the Mac. The software version in this image is 1.6.5, but the process is exactly the same for newer versions.

When the extraction completes, you will have a new file, which is the actual IDE. All you have to do then is to move it into your Applications folder.



The IDE application is now in my Downloads folder.



... And finish the process by copying the file into your Applications folder.

To start the IDE, double-click on the Arduino icon inside the Applications folder.

How to install the Arduino IDE on Windows

On Windows, start by double-clicking on the installation file. Be ready for a long series of confirmation dialogue boxes. In all of them, it is safe to accept the defaults.

A pop-up will ask you for permission to run the program, click on Yes to continue:



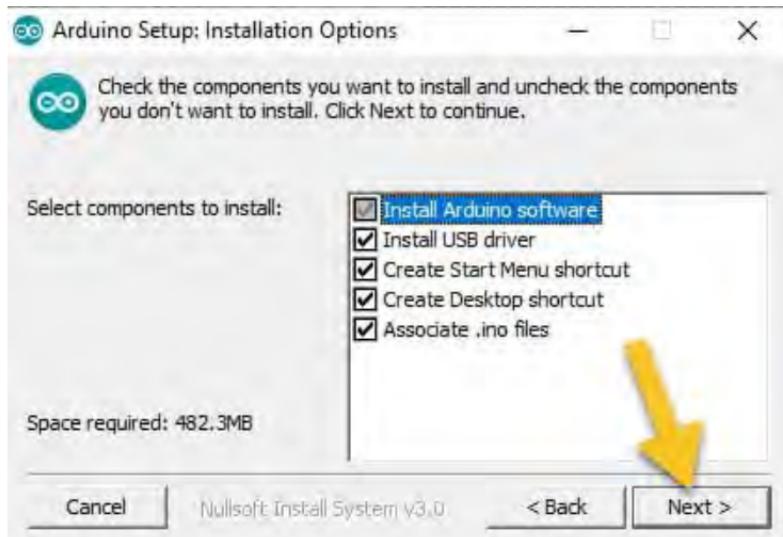
Yes! It is safe to continue!

Agree to the license agreement:



Yes! I agree...

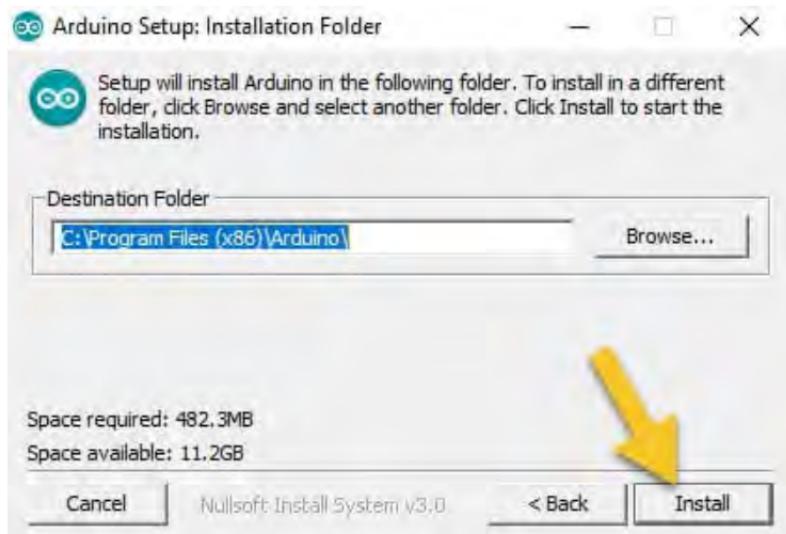
Accept the checked components:



Yes! All these components are used.

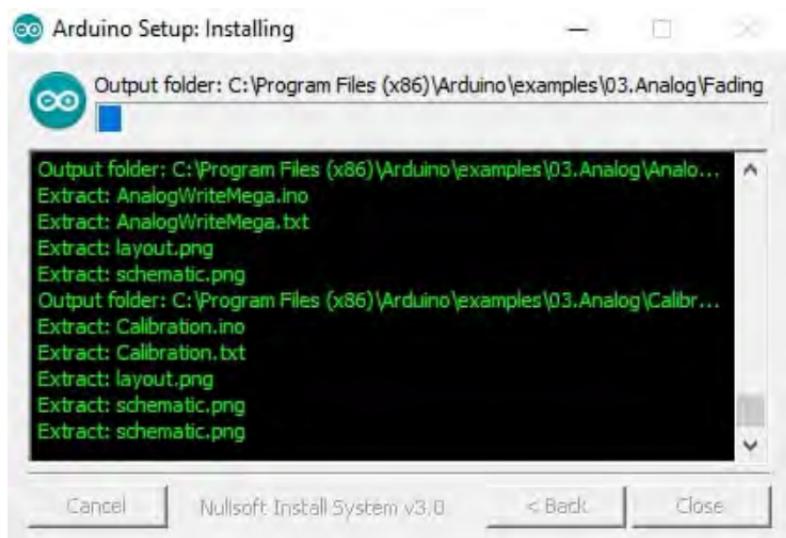
Accept the default installation location, and the file copy will

begin:



Yes! This location looks fine.

The installer will start copying files to the installation location:



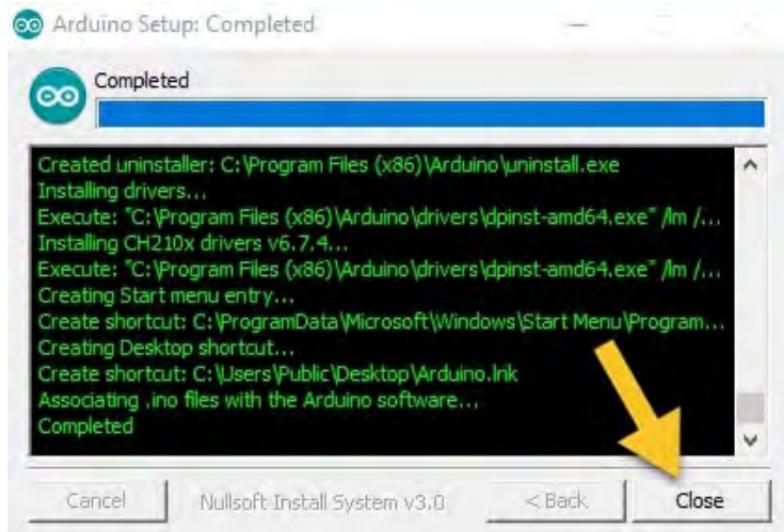
Copying files to the destination.

You may be asked to install device drivers a couple of times.
Click on “Install” to accept the installation.



Yes, these USB device drivers are also useful.

When the file copy process finishes, click “Close” to close the installer:



When the file copy process finishes, click “Close” to close the installer:

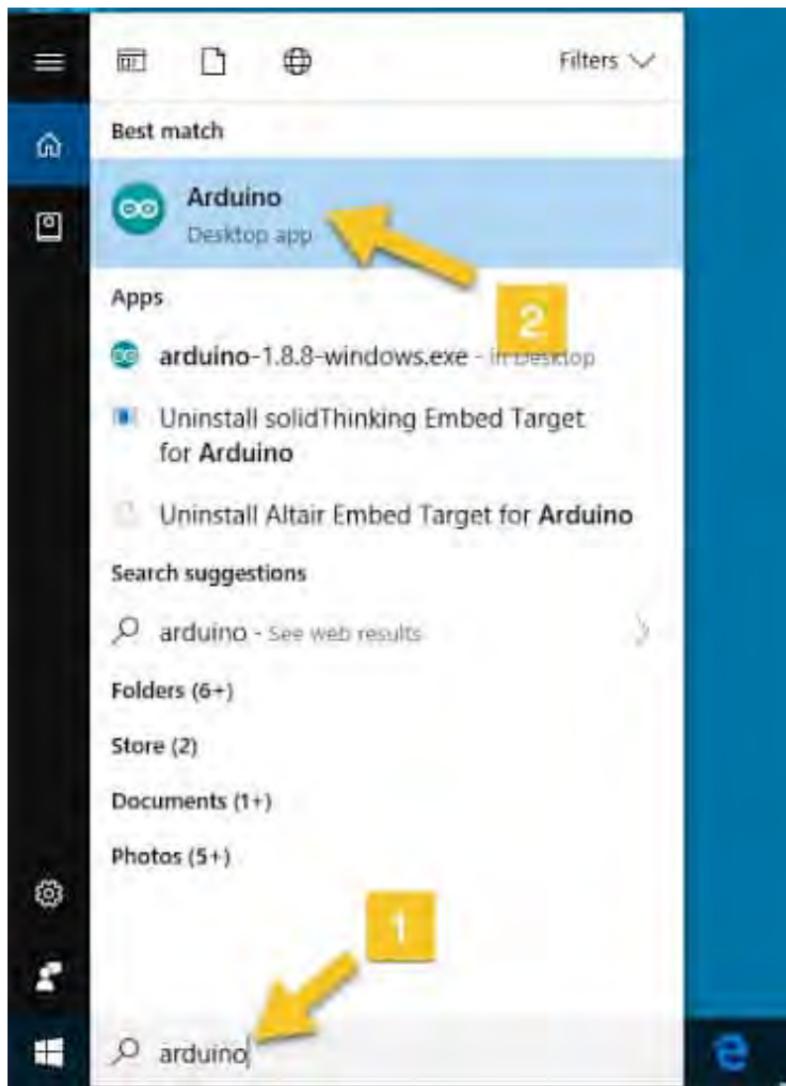
You can start the Arduino IDE, just like any other Windows program. If you accepted the default installation options, then you should be able to find the Arduino shortcut on the desktop (see screenshot example below).

Double-click on the shortcut to start the IDE.



The Arduino IDE shortcut on the desktop

An alternative way to start the IDE is to search for it. In Windows 10, you will find the search field in the bottom left corner of the desktop. Type “Arduino” in the field, and you will see the “Arduino” application appearing at the top of the search results (see the example screenshot below. Just click on the application name to start the IDE.



You can start the Arduino IDE by searching for it.

Awesome. Now that you have installed the Arduino IDE, you can use it to upload your first sketch.

Let's try it out.

Upload your first sketch

Start the Arduino IDE. A few seconds later, you should see a new IDE window with a blank sketch. It will look like the example below (Windows 10 and Mac OS):

The screenshot shows the Arduino IDE interface on a Windows 10 system. The title bar reads "Blink | Arduino 1.8.8". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with icons for Open, Save, Print, and others. The main workspace is titled "Blink" and contains the following code:

```
25 // the setup function runs once when you press reset or power
26 void setup() {
27     // initialize digital pin LED_BUILTIN as an output.
28     pinMode(LED_BUILTIN, OUTPUT);
29 }
30
31 // the loop function runs over and over again forever
32 void loop() {
33     digitalWrite(LED_BUILTIN, HIGH);      // turn the LED on (HIGH)
34     delay(1000);                      // wait for a second
35     digitalWrite(LED_BUILTIN, LOW);     // turn the LED off by

```

At the bottom of the IDE, it says "Arduino/Genuino Uno on COM3".

A blank Arduino sketch (Windows 10)

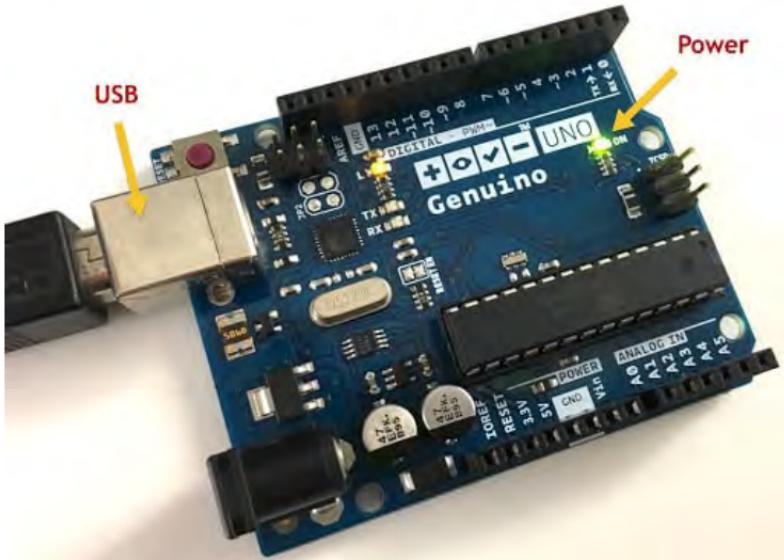
The screenshot shows the Arduino IDE interface on a Mac OS X system. The title bar reads "sketch_feb13a | Arduino 1.8.8". Below the title bar is a toolbar with various icons. The main area contains the following code:

```
void setup() {  
    // put your setup code here, to run once:  
}  
  
void loop() {  
    // put your main code here, to run repeatedly:  
}
```

The bottom right corner of the code editor window displays "Arduino/Genuino Uno".

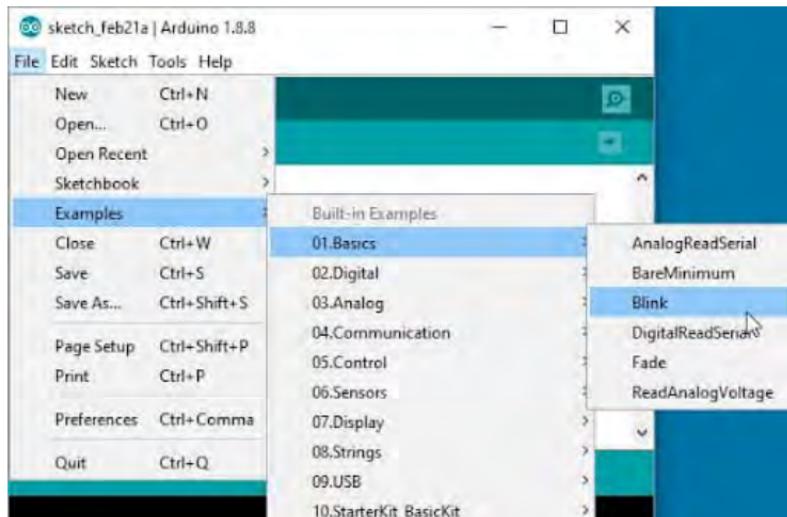
A blank Arduino sketch (Mac OS)

Continue to connect your Arduino to your computer via the USB cable. When connected, the power LED on the Arduino should light up.



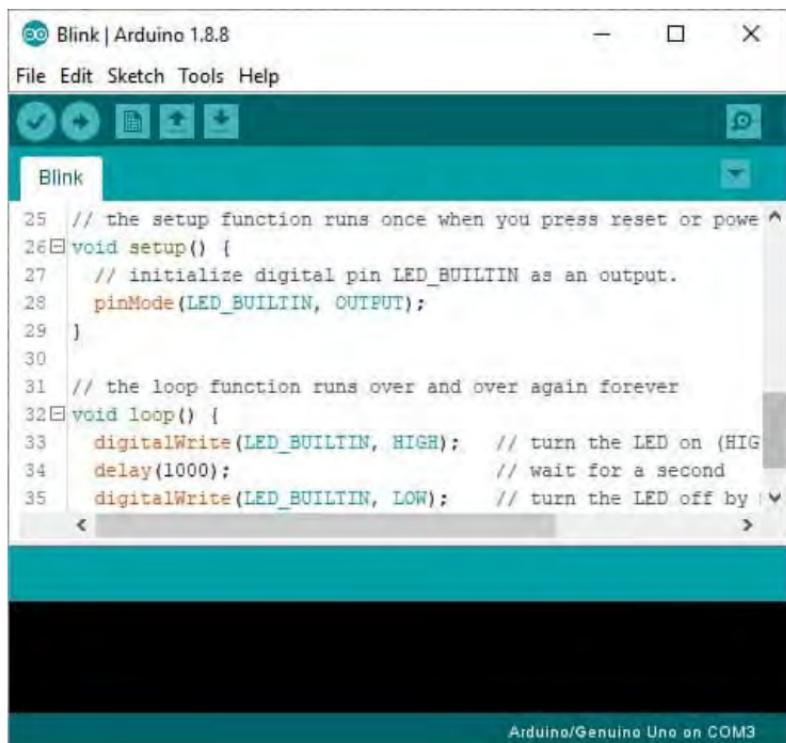
The Arduino is connected to your computer via a USB cable, and the power LED is lit

Go back to the Arduino IDE and load one of the examples. On both Windows 10 and Mac OS, you can do this by clicking on File, Examples, 01.Basics, Blink.



Load the Blink sketch on Windows 10

The Blink sketch will appear. Don't worry about what is in it, since you will learn about programming (sketching) in the next two lessons. For now, it's enough to know that this little program will make the LED marked "L" on the Arduino Uno to blink once every second. This LED is right next to the pin marked 13.



A screenshot of the Arduino IDE interface. The title bar says "Blink | Arduino 1.8.8". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with icons for upload, refresh, and file operations. The main area shows the "Blink" sketch code:

```
25 // the setup function runs once when you press reset or power
26 void setup() {
27     // initialize digital pin LED_BUILTIN as an output.
28     pinMode(LED_BUILTIN, OUTPUT);
29 }
30
31 // the loop function runs over and over again forever
32 void loop() {
33     digitalWrite(LED_BUILTIN, HIGH);      // turn the LED on (HIGH)
34     delay(1000);                      // wait for a second
35     digitalWrite(LED_BUILTIN, LOW);     // turn the LED off by

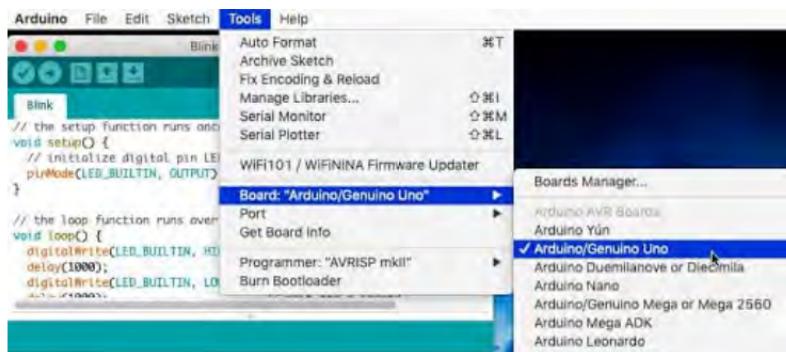
```

The status bar at the bottom right says "Arduino/Genuino Uno on COM3".

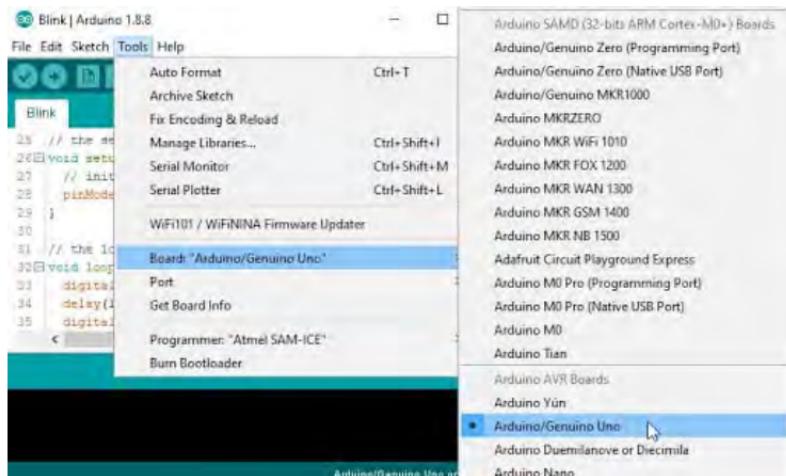
The Blink sketch

Before you can upload this sketch to your Arduino, there are two things you should check and set. The Arduino board model, and the port to which it is connected. Both are usually automatically and correctly set by the Arduino IDE, but I have a habit of checking before I upload the sketches, that has saved me a huge amount of time over the years.

To check for the correct board model, click on Tools, Board, and Arduino/Genuino Uno.

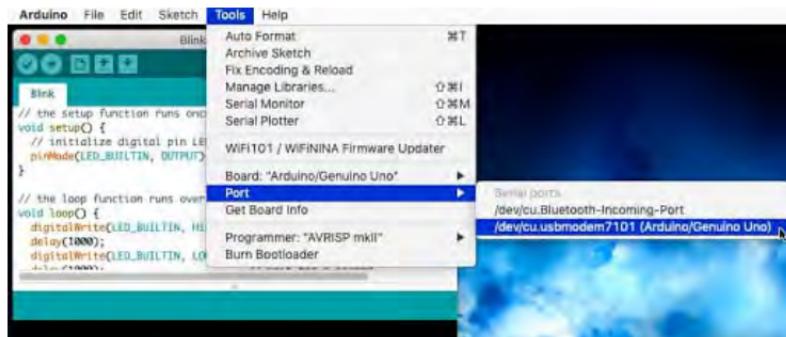


Select the Arduino Uno Board (Mac OS)

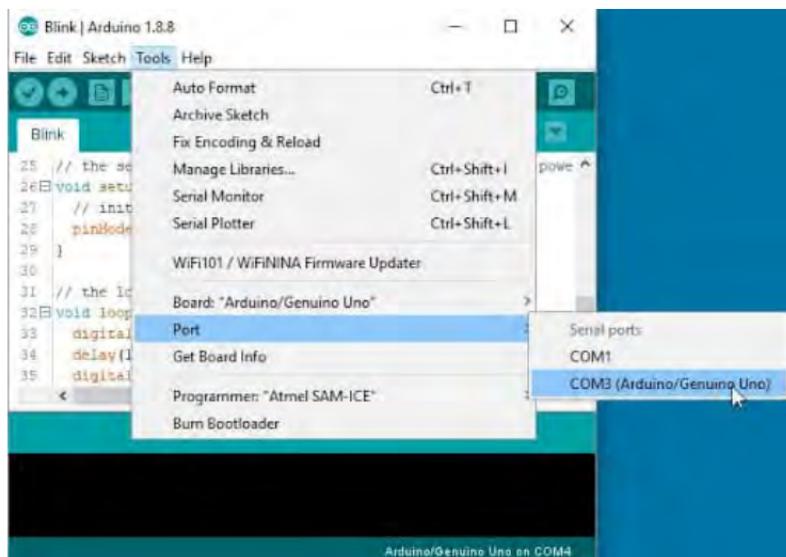


Select the Arduino Uno Board (Windows 10)

Next, check the port. Click on Tools, Port, /dev/cu/usbmodem7101 (Arduino/Genuino Uno) or COM3. The port names may vary, but the name of the board should also appear on the menu to help you identify the correct one.



Select the correct port (Mac OS)



Select the correct port (Windows 10)

With the settings verified, go ahead to upload. Click on the icon with the right arrow to start the upload. When you click on this button, the button will turn yellow, and the IDE will first compile and then upload the sketch to your connected Arduino. This process takes around 10 seconds.

The screenshot shows the Arduino IDE interface. The title bar reads "Blink | Arduino 1.8.8". The toolbar includes icons for file operations like Open, Save, and Upload. The main window displays the "Blink" sketch code:

```
// the setup function runs once when you press reset or power the board
void setup() {
    // initialize digital pin LED_BUILTIN as an output.
    pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
    digitalWrite(LED_BUILTIN, HIGH);      // turn the LED on (HIGH is the
    delay(1000);                         // wait for a second
    digitalWrite(LED_BUILTIN, LOW);       // turn the LED off by making it
    delay(1000);                         // wait for a second
```

Below the code, a progress bar indicates the upload process. A status message at the bottom left says "Uploading...". Another message below it provides memory usage details:

Sketch uses 930 bytes (2%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes free.

At the bottom right, the connection information is shown: "Arduino/Genuino Uno on /dev/cu.usbmodem7101".

Uploading a sketch (Mac OS)

The screenshot shows the Arduino IDE interface. The title bar says "Blink | Arduino 1.8.8". The menu bar includes File, Edit, Sketch, Tools, and Help. The toolbar has icons for file operations like Open, Save, and Print. A search bar at the top says "Blink". The code editor contains the standard "Blink" sketch:

```
25 // the setup function runs once when you press reset or power
26 void setup() {
27     // initialize digital pin LED_BUILTIN as an output.
28     pinMode(LED_BUILTIN, OUTPUT);
29 }
30
31 // the loop function runs over and over again forever
32 void loop() {
33     digitalWrite(LED_BUILTIN, HIGH);      // turn the LED on (HIGH)
34     delay(1000);                      // wait for a second
35     digitalWrite(LED_BUILTIN, LOW);     // turn the LED off by :
```

The status bar at the bottom shows "Compiling sketch" with a progress bar, the command line "C:\Program Files (x86)\Arduino\arduino-builder -dump-prefs -logger=native", and the port information "Arduino/Genuino Uno on COM3".

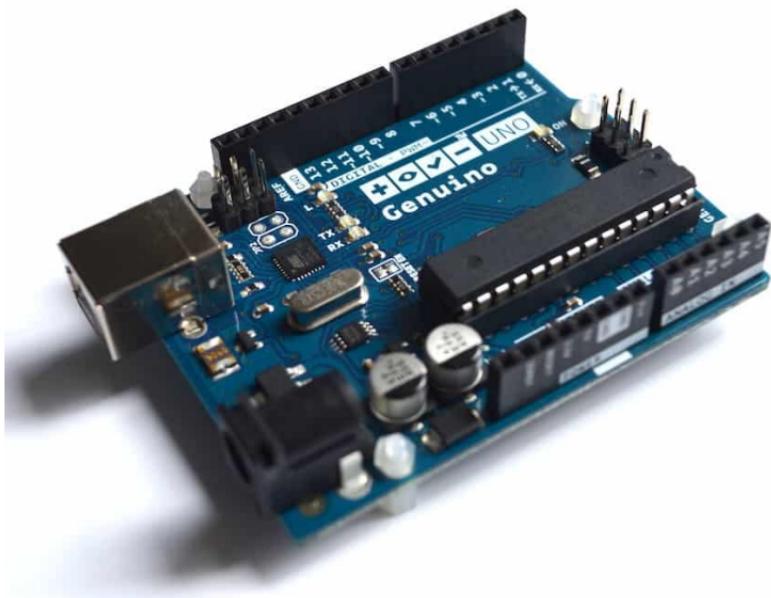
Uploading a sketch (Windows 10)

When the process finishes, have a look at the LED marked "L," next to pin 13. Can you verify that it is blinking once per second? It's the sketch you just uploaded that made this happen.

And this is how easy it is. But it's only the start.

In the next lesson, you'll take an in-depth look at Arduino libraries, which help make programming a breeze.

When you are ready, [continue with the next lesson](#). There, you will learn about one of Arduino's superpowers: libraries.



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

Lesson 5: Arduino libraries and how to install them

Introduction to the Arduino guide series

Arduino libraries and how to install them

In Lesson 5 of the introductory mini-course on the Arduino, you will learn about Arduino libraries.

A library is a software code that performs a particular function. It is distributed by its author so that other people that need this function can include it in their applications without having to write the equivalent code again.

In Lesson 5 of the introductory mini-course on the Arduino, you will learn about Arduino libraries.

A library is a software code that performs a particular function. It is distributed by its author so that other people that need this function can include it in their applications without having to write the equivalent code again.

Libraries can speed up and simplify prototyping significantly. When you install the Arduino programming environment on your computer, you also install a lot of frequently used libraries that you can use right away.

If there is a library that you need but is not included with the IDE, you can install it. We'll look at an example shortly.

But first, why are libraries important?

Why libraries?

It's simple: libraries allow you to be far more productive and effective because they make it possible for you to use existing, tested, high-quality code.

Imagine you are a mechanic, or a builder, or a tailor. As a mechanic, when you need a new part, you can either design it and create it yourself (a long, expensive, error-prone process) or order it from a spare parts vendor. If you are a builder and need bricks, you can either make them yourself (a long, expensive, error-prone process) or order them from the brickyard. If you are a tailor and you need some new fabric, you can either plant some cotton, harvest it, and produce your fabric out of it, or just buy it from the market.

As Makers, our objective is to create new things, things that solve a particular problem we have, or help us learn. The tools that we use, combined with our skill and knowledge, dictate how effective we will be in our making.

The Arduino is extraordinary in the technology world because of the abundance of shared and reusable code. Thanks to the Arduino libraries, it is possible for us to build our projects on the shoulders of other makers. This can make, even a beginner, very productive.

This is why I decided it is important to introduce you to libraries now, instead of waiting for a later time.

Let's look at an example.

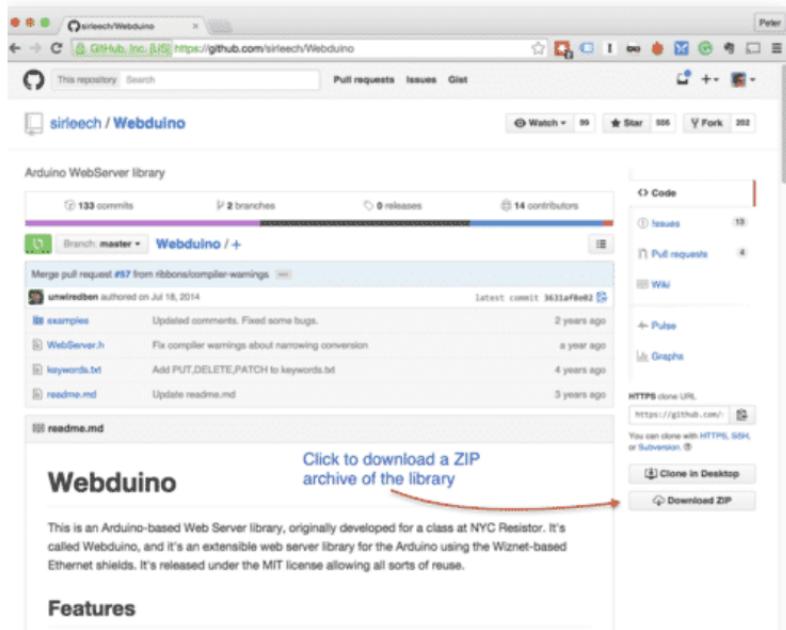
An example of a library and how to install it

Let's say that you want to have a small web server running on your Arduino. You can set up this server so that you can use your browser to control lights and read sensor values

connected to it. The Arduino can handle this, no problem. You could spend a few days (or weeks) and write your own bare-bones web server (assuming you have a good understanding of HTTP), or use Webduino.

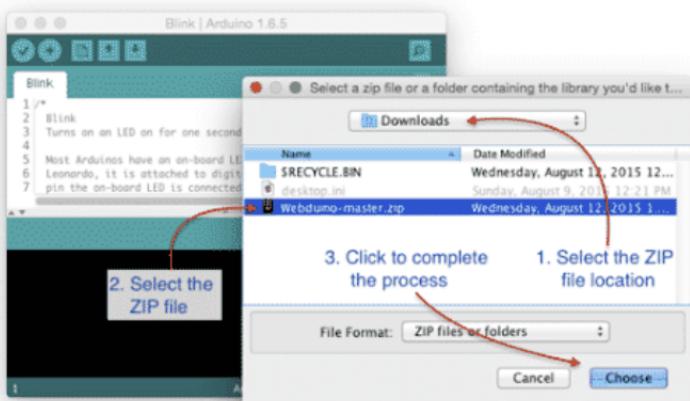
Webduino is a library that was written at NYC Resistor to make it very easy to turn an Arduino into a basic web server.

The home page for Webduino on Github is
<https://github.com/sirleech/Webduino>.



Including a new library

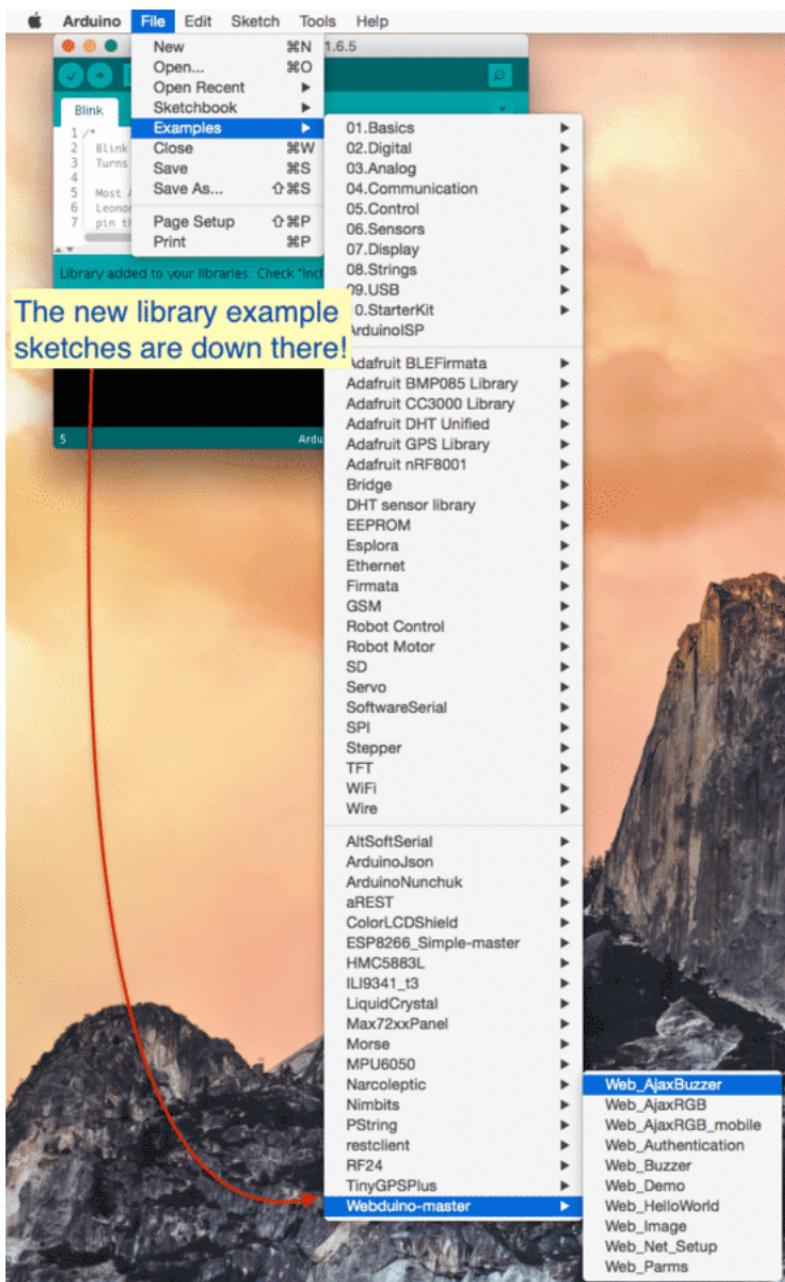
A new dialogue box will pop up. Browse to the location of the ZIP file, select it, and click on Choose to complete the process:



The Library addition dialogue box

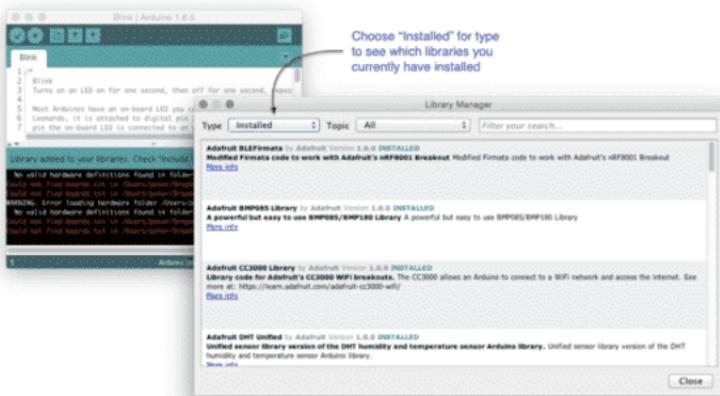
When you click on “Choose,” the dialogue box will disappear, but nothing else is going to happen. No confirmation, no sound. To make sure that the Webduino library was installed, you can look for the example sketches that most libraries include.

Go to File Examples, and look at the bottom of the list for your new library:



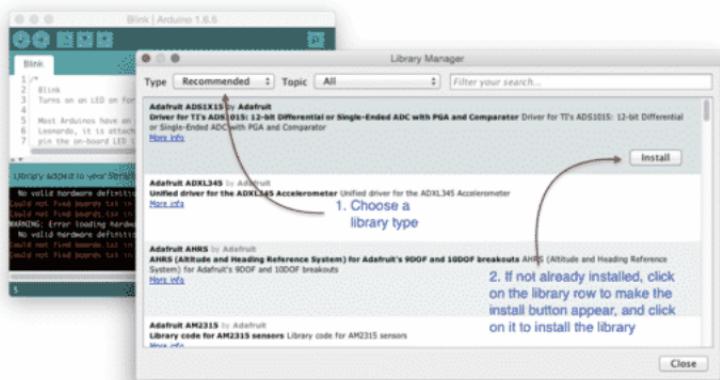
The new library example sketches are down there!

You can also find a list of names and descriptions of all the libraries currently installed in your IDE. Go to Sketch Include Library Manage Libraries, and this window will pop-up:



The library manager can tell you what's installed and install new libraries.

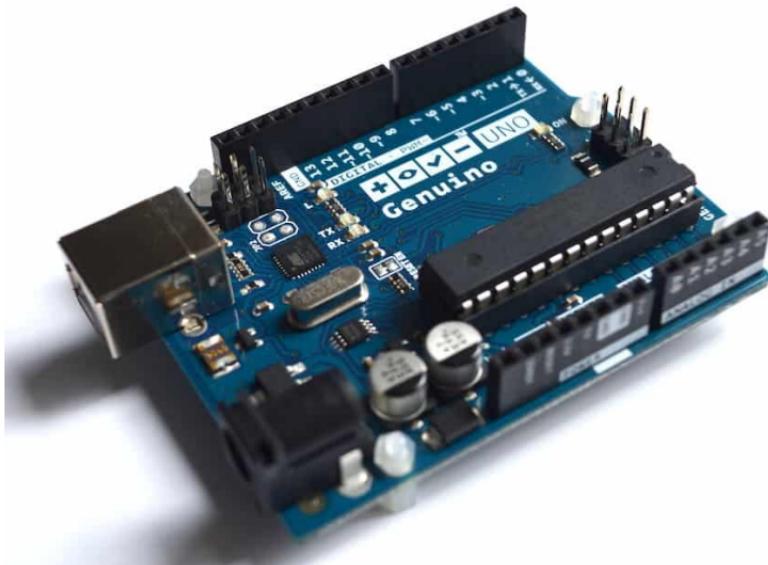
The Library Manager, apart from telling you what is already installed, can also install new libraries from online sources with the click of a button.



You can add a new library from the Library Manager.

Now you should have a good overview of the IDE and its most essential functions. Let's have a look at the Arduino programming language next.

Ready for a bit of programming? [Continue with lesson 6.](#)



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

Lesson 6: The basics of Arduino programming: program structure, functions, variables, operators

Introduction to the Arduino guide series

The basics of Arduino programming: program structure, functions, variables, operators

In this lesson, we discuss the basics of Arduino programming to help you understand the basic concepts of the Arduino language: the structure, the functions, the variables and the operators.

On the second-last lesson of our 7-lesson introduction course on the Arduino, we're going to discuss the basics of Arduino programming.

In the [previous lesson](#), you learned about the power of the libraries that are part of the Arduino ecosystem, and how these libraries can help turbo-boost your productivity.

Just like a builder needs to know how to use the brick and other components that are used in building a new house, similarly, you will need to know how to use and extend the libraries.

Even more important, you will need to know how to write your own code.

The combination of

1. understanding the basics of programming,
2. understanding the basics of electronics,
3. and understanding how to use the shared work of others,

will make it possible for you to create amazing things.

This is what this (and the next and last lesson) is about: helping you understand the basic concepts in programming the Arduino.

The Arduino language

The Arduino language is C++.

Most of the time, people will use a small subset of C++, which looks a lot like C. If you are familiar with Java, then you will find C++ easy to work with and to recognize. If you have never programmed before, do not worry, and do not be afraid. In the next few paragraphs, you will learn everything you need to get started.

The most important “high level” characteristic of C++ is that it is object-oriented. In such a language, an object is a construct that combines functional code (the code that does things like calculations and memory operations), with “state” (the results of such calculations, or simply values, stored in variables).

Object orientation made programming much more productive in most types of applications when compared with earlier paradigms because it allowed programmers to use abstractions to create complicated programs.

For example, you could model an Ethernet adaptor as an object that contains attributes (like its IP and MAC addresses) and functionality (like asking a DHCP server for network

configuration details). Programming with objects became the most common paradigm in programming, and most modern languages, like Java, Ruby, and Python, have been influenced heavily by C++.

Much of the sketch code you will be writing and reading will be referencing libraries containing definitions for objects (these definitions are called “classes”). Your original code, to a large extent, will consist of “glue” code and customizations. This way, you can be productive almost right away by learning a small subset of C++.

The code that makes up your sketch must be compiled into the machine code that the microcontroller on the Arduino can understand. This compilation is done by a special program, the compiler. The Arduino IDE ships with an open-source C++, so you don’t have to worry about the details. Imagine: every time you click the “Upload” button, the IDE starts up the compiler, which converts your human-readable code into ones and zeros, and then sends it to the microcontroller via the USB cable.

As every useful programming language, C++ is made up of various keywords and constructs. There are conditionals, functions, operators, variables, constructors, data structures, and many other things.

In this lesson, you will learn about the structure of an Arduino program, functions, and variables. Take a bit of time to consolidate this new knowledge, because next, you will complete this series with the last lesson in which you will learn how to program your Arduino to make decisions, and interact with the outside world.

Let’s take the most important of those things to examine them one at a time.

The structure of an Arduino sketch

The simplest possible Arduino sketch is this (click here to see the Gist for this sketch):

```
void setup() { // put your setup code here, to run once:}void  
loop() { // put your main code here, to run repeatedly:}
```

This code contains two functions in it.

The first one is **setup()**. Anything you put in this function will be executed by the Arduino just once when the program starts.

The second one is **loop()**. Once the Arduino finishes with the code in the **setup()** function, it will move into a **loop()**, and it will continue running it in a loop, again and again, until you reset it or cut off the power.

Notice that both **setup()** and **loop()** have open and close parenthesis? Functions can receive parameters, which is a way by which the program can pass data between its different functions. The setup and loop functions don't have any parameters passed to them. If you add anything within the parenthesis, you will cause the compiler to print out a compilation error and stop the compilation process.

Every single sketch you write will have these two functions in it, even if you don't use them.

In fact, if you remove one of them, the compiler again will produce an error message. These are two of the few expectations of the Arduino language.

These two functions are required, but you can also make your own. Let's look at this next.

Custom functions

A function is merely a group of instructions with a name. The Arduino IDE expects that the **setup()** and **loop()** functions will be in your sketch, but you can make your own. Group instructions inside functions is a good way of organizing your sketches, especially as they tend to get bigger in size and complexity as you become a more confident programmer.

To create a function, you need a definition and the code that goes inside the curly brackets.

The definition is made up of at least:

- a return type
- a name
- a list of parameters

Here's an example

```
int do_a_calc(int a, int b){ int c = a + b; return c;}
```

The return type here is **int** in the first line. It tells the compiler that when this function finishes its work, it will return an integer value to the caller (the function that called it).

The name (also known as the “identifier”) of the function is **do_a_calc**. You can name your functions anything you like as long as you don't use a reserved word (that is, a word that the Arduino language already uses), it has no spaces or other special characters like %, \$ and #. You can't use a number as the first character. If in doubt, remember only to use letters, numbers, and the underscore in your function names.

In the first line of the body, we create a new variable, **c**, of type integer (**int**). We add **a** and **b** and then assign the result to **c**.

And finally, in the second line of the body of the function, we

return the value stored in **c** to the caller of **do_a_calc**.

Let's say that you would like to call **do_a_calc** from your setup function. Here's a complete example showing how to do that:

```
void setup(){ // put your setup code here, to run once: int a =  
do_a_calc(1,2);}  
void loop(){ // put your main code here, to run  
repeatedly:}  
int do_a_calc(int a, int b){ int c = a + b; return c;}
```

In the **setup()** function, the second line defines a new variable, **a**. In the same line, it calls the function **do_a_calc**, and passes integers 1 and 2 to it. The **do_a_calc** function calculates the sum of the two numbers and returns the value 3 to the caller, which is the second line of the **setup()** function. Then, the value 3 is stored in variable **a**, and the **setup()** function ends.

There's a couple of things to notice and remember.

Comments

Any line that starts with `//` or multiple lines that start with `/*` and finish with `*/` contain comments.

Comments are ignored by the compiler. They are meant to be read by the programmer.

Comments are used to explain the functionality of code or leave notes to other programmers (or to self).

Scope

In the **setup()** function, there is a definition of a variable with an identifier **a**. In function **do_a_calc**, there is also a definition of a variable with the same identifier (it makes no difference that this definition is in the function definition line).

Having variables with the same name is not a problem as long

as they are not in the same scope. A scope is defined by the curly brackets. Any variable between an open and close curly bracket is said to be within that scope. If there is a variable with the same name defined within another scope, then there is no conflict.

Be careful when you choose a name for your variables. Problems with scopes can cause headaches: you may expect that a variable is accessible at a particular part of your sketch, only to realize that it is out of scope.

Also, be careful to use good descriptive names for your variables. If you want to use a variable to hold the number of a pin, call it something like:

```
int digital_pin = 1; instead of int p = 1;
```

You will thank yourself later.

Variables

Programs are useful when they process data. Processing data is what programs do, all the time. Programs will either get some data to process from a user (perhaps via a keypad). From a sensor (like a thermistor that measures temperature), the network (like a remote database), a local file system (like an SD Card), a local memory (like an EEPROM), and so many other places.

Regardless of the place where your program gets its data from, it must store them in memory to work with it. To do this, we use variables. A variable is a programming construct that associates a memory location with a name (an identifier). Instead of using the address of the memory location in our program, we use an easy to remember a name. You have already met a variable. In the earlier section on custom functions, we defined a bunch of variables, **a**, **b** and **c**, that each holds an integer.

Variables can hold different kinds of data other than integers. The Arduino language (which, remember, is C++) has built-in support for a few of them (only the most frequently used and useful are listed here):

C++ Keyword	Size	Description
boolean	1 byte	Holds only two possible values, true or false , even though it occupies a byte in memory.
char	1 byte	Holds a number from -127 to 127. Because it is marked as a “char,” the compiler will try to match it to a character from the ASCII table of characters .
byte	1 byte	Can hold numbers from 0 to 255.
int	2 byte	Can hold numbers from -32768 to 32767.
unsigned int	2 byte	Can hold numbers from 0-65535
word	2 byte	Same as the “unsigned int.” People often use “word” for simplicity and clarity.
long	4 byte	Can hold numbers from -2,147,483,648 to 2,147,483,647.
unsigned long	4 byte	Can hold numbers from 0-4,294,967,295
float	4 byte	Can hold numbers from -3.4028235E38 to 3.4028235E38. Notice that this number contains a decimal point. Only use float if you have no other choice. The ATMEGA CPU does not have the hardware to deal with floats, so the compiler has to add a lot of code to make it possible for your sketch to use them, making your sketch larger and slower.

string -	-	A way to store multiple characters as an array of chars. C++ also offers a String object that you can use instead that provides more flexibility when working with strings in exchange for higher memory use.
array	-	A structure that can hold multiple data of the same type.

To create a variable, you need a valid name and a type. Just like with functions, a valid name is one that contains numbers, letters, and an underscore starts with a letter and is not reserved. Here is an example:

```
byte sensor_A_value;
```

This line defines a variable named **sensor_A_value**, which will hold a single byte in memory. You can store a value in it like this:

```
sensor_A_value = 196;
```

You can print out this value to the serial monitor like this:

```
Serial.print(sensor_A_value);
```

The serial monitor is a feature of the Arduino IDE that allows you to get a text from the Arduino displayed on your screen. More about this later, here I want to show you how to retrieve the value stored in a variable. Just call its name. Also, remember the earlier discussion about scope: the variable has to be within scope when it is called.

Another beautiful thing about a variable is that you can change the value stored in it. You can take a new reading from the sensor and update the variable like this:

```
sensor_A_value = 201;
```

No problem, the old value is gone, and the new value is stored.

Constants

If there is a value that will not be changing in your sketch, you can mark it as a constant.

Constants have benefits regarding memory and processing speed, and it is a good habit to use these.

You can declare a constant like this:

```
const int sensor_pin = 1;
```

Here, you define the name of the variable **sensor_pin**, mark it as constant, and set it to 1. If you try to change the value later, you will get a compiler error message, and your program will not even get uploaded to the Arduino.

Operators

Operators are special functions that operate one or more pieces of data.

Most people are familiar with the basic arithmetic functions, = (assignment), +, -, * and /, But there are a lot more.

For example, here are the most commonly used operators:

Operator	Function	Example
%	Modulo operator. It returns the remainder of a division.	$5 \% 2 = 1$
$+=$, $-=$, $*=$, $/=$	Compound operator. It performs an operation on the current value of a variable.	<code>int a = 5; a += 2;</code> This will result in a containing 7 (the original 5 plus a 2 from the addition operation).

Operator	Function	Example
<code>++, --</code>	Increment and decrement by 1.	<code>int a = 5;a++;</code> This will result in a becoming 6.
<code>==, !=, <, >, <=, >=</code>	Comparison operators. Will return a boolean (true or false) depending on the comparison result. <ul style="list-style-type: none">•• <code>==</code> equality•• <code>!=</code> un-equality•• <code><</code> less than•• <code>></code> greater than•• <code><=</code> less or equal than•• <code>>=</code> greater or equal than	<code>int a = 5;int b = 6;boolean c = a == b;</code> This will result in variable c contains a false boolean value.
<code>!, &&, </code>	Logical operators. The “!” operator will invert a boolean value. ! NOT (invert) of a boolean value && AND of two booleans OR of two booleans	<code>boolean a = true;boolean b = true;boolean c = false;boolean x = !a; // x falseboolean y = b && c; // y falseboolean z = b c; // z true</code>

There are more than these. If you want to work at the bit level, for example, and manipulate individual bits within a byte (useful for things like shift registers), you can use bitwise operators. But this is something you can pick up and learn

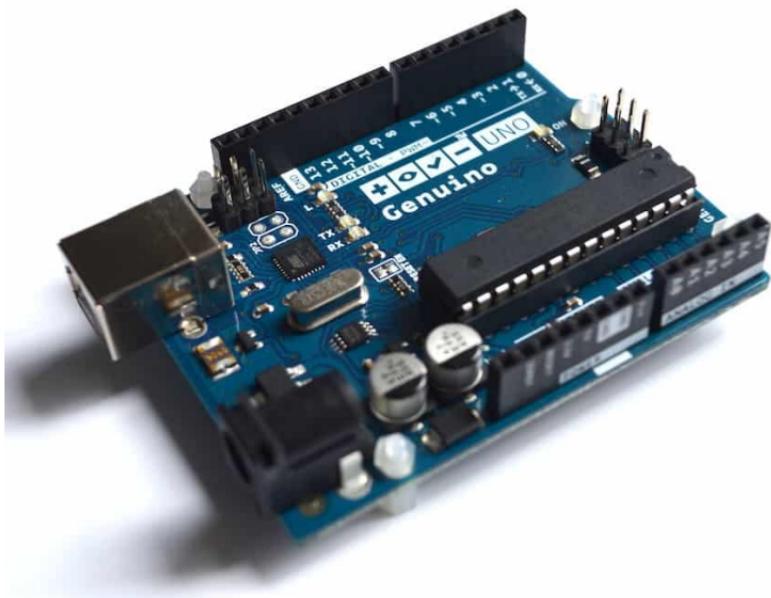
later.

Ok, that is enough for one lesson.

I hope that now, you have a clear understanding of these fundamental concepts in programming: the structure of an Arduino program, functions, variables, and operators. Just with what you have learned so far, you can write very interesting simple programs.

But the real fun begins when your program can make decisions, and when these programs can sense and control the environment. You will learn about these capabilities in tomorrow's lesson.

There's only one lesson left in this series, in which you will learn about Loops, conditions, objects, inputs & outputs. [Would you like to continue?](#)



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

Lesson 7: The basics of Arduino programming: Loops, conditions, objects, inputs & outputs

Introduction to the Arduino guide series

The basics of Arduino programming: Loops, conditions, objects, inputs & outputs

In this lesson, we discuss the basics of Arduino programming to help you understand the basic concepts of the Arduino language: loops and conditionals, classes and objects, inputs and outputs.

This is the last article of the **Getting Started** series. We're going to complete our discussion of the basics of Arduino programming.

In the [previous lesson](#), you learned about things like variables, functions, and loops.

Today, you will build on this knowledge and learn three new important concepts:

1. The programming structures that allow your Arduino (and any computer) to make decisions and repeat instructions.
2. Classes and objects that allow us (the

programmers) to create reliable programs that resemble concepts from our real-world experiences.

3. Inputs and outputs that enable us to connect external components like buttons and lights, to the Arduino.

Let's begin with loops (structures we use to repeat instructions) and conditionals (structures we use for decision making).

Loops and conditionals

Conditionals are useful when you want to change the flow of executing in your sketch. Loops are useful when you want to repeat a block of code multiple times.

Very often, these two work together; that's why I discuss them here in the same section.

Let's start with a conditional. Imagine you have a red light and a green light. You want to turn the green light on when you press a button and the red light on when you leave the button not pressed.

To make this work, you can use a conditional.

conditional: "if..else"

The most common of these is the **if...else** statement. Using pseudo code (that is, a program written in English that looks a bit like a real program), you would implement this functionality like this:

```
if (button == pressed){green_light(on);red_light(off);}
else{red_light(on);green_light(off)}
```

loop: “while”

If you need to repeat a block of code based on a boolean condition, you can use the while conditional expression. For example, let's say that you want to make a noise with a buzzer for as long as you press a button. Using pseudo code again, you can do it like

```
this:while(button_is_pressed){make_annoying_noise;}
```

Easy!

loop: “do_while”

You can do the same thing but do the check of the condition at the end of the block instead of the start. This variation would look like this:

```
do{make_annoying_noise;} while(button_is_pressed)
```

loop: “for”

If you know how many times you want to repeat code in a block, you can use the **for** structure. Let's say you want to blink a light 5 times.

Here's how to do it:

```
for (n = 1 to 5){Turn light on;Turn light off;}
```

Your light will turn on and then off 5 times. Inside the curly brackets, you will also have access to the **n** variable, which contains the number of a repeat at any given time. With is, you could insert a conditional so that you leave the lights on before the last loop ends:

```
for (n = 1 to 5){Turn light on;if (n < 5) then Turn light off;}
```

In this variation, the light will only turn off if the **n** variable is

less than 5.

conditional: “switch”

Another useful conditional is the **switch**. If you have a variable, like **button_pressed**, that can take a few valid values; you can do something like this with it:

```
switch (button_pressed){case 1:Blink light one  
time;break;case 2:Blink light two times;break;case 3:Blink light  
three times;break;default:Don't blink light;}
```

The **switch** statement will check the value stored in the **button_pressed** variable. If it is 1, it will blink the light once, if it is 2, it will blink the light twice, and if it is 3, it will blink three times. If it is anything else, it won't blink the light at all (this is what the “default” case is).

The **button_pressed** variable can be an integer and could be taking its values from a membrane keypad, like this one:



A membrane keypad can be used to provide input to your sketch.

For now, don't worry about how this keypad works; this is something you will learn later. Just imagine that when you hit a key, a number comes out.

Also, notice the keyword "break"? It will cause the execution of the sketch to jump out of the block of code that is between the curly brackets. If you remove all the "break" statements from your sketch and press 1 on the keypad, then the sketch will cause the light to blink once, then twice, and then three times as the execution will start in the first case clause, and then move into the rest.

Classes and objects

You now know that the Arduino language is actually C++ with a lot of additional support from software, the libraries which were mentioned earlier, that makes programming easy. It was also said that C++ is an object-oriented programming language.

Let's have a closer look at this feature and especially how it looks like in Arduino code.

Object-orientation is a technique for writing programs in a way that makes it easier to manage as they grow in size and complexity. Essentially, a software object is a model of something that we want the computer (or an Arduino) to be able to handle programmatically.

Let me give you an example. Imagine that you have a robotic hand. The arm only has one finger and can rotate by 360 degrees. The finger can be open or closed. You can model this hand in an object-oriented way like in this pseudo-code:

```
class robotic_hand{//These variables hold the state of the  
handbool finger;int rotation;//These variables change the state  
of the handfunction open_finger();function  
close_finger();function rotate(degrees);//These variables report  
the state of the handfunction bool  
get_finger_position();function int get_rotation_position();}
```

Can you understand what this code does? I am creating a model of the hand and giving it the name **robotic_hand**. The keyword "class" is a special keyword so that the compiler understands my intention to create a model.

Inside the class, I define three kinds of components for the model (=class). First, a couple of variables to hold the current state of the hand. If the hand is in an open position, the boolean variable **finger** will be **true**. If the hand is rotated at 90 degrees, the integer variable **rotation** will contain 90.

The second set of components are special functions that allow me to change the status of the hand. For example, if the hand is currently open and I want to close it so that it can pick up an object, I can call the **close_finger()** function. If I want to rotate it at 45 degrees, I can call **rotate(45)**.

Finally, the third set of components are functions that allow me to learn about the status of the hand. If I want to know if the hand is opened or closed, I can call **get_finger_position()**, and this function will respond with **true** or **false**.

The names are up to me to choose so that their role is clear. A class hides within its components such as these, so the programmer can think more abstractly about the thing they are working with, instead of the implementation details.

Let's say now that you would like to use this class in your sketch. Here is an example of how you would do it in Arduino:

```
#include <Robot_hand.h>Robot_hand robot_hand();void  
setup(){}  
loop(){robot_hand.open_finger();robot_hand.rotate(45);robot_h  
and.close_finger();}
```

You would start by importing the **Robot_hand library**, which contains the class you just created into your Arduino sketch. You do this with the include statement in the first line of your sketch.

In the second line, you create an object based on the **Robot_hand** class. Think about this for a few moments: a class contains the blueprints of an object but is not an object; it is the equivalent of a blueprint for a house, and the house itself. The blueprint is not a house, only the instructions for building a house. The builder will use the blueprint as the instructions to build a house.

Similarly, the robot hand class definition is only the instructions that are needed for building the robot hand object

in your sketch. In the second line of this example sketch, we are defining a new object build based on the instructions in the **Robot_hand** class, and we give it the name **robot_hand()**. The object's name cannot be the same as the name of the class, that is why it starts with a lowercase r.

In the **loop()** function, we can call the object's functions to make the robot hand move. We can open it using **robot_hand.open_finger()** and close it using **robot_hand.close_finger()**. Notice that these instructions start with the name of the object, **robot_hand**, followed by a dot, then followed by the name of the function we want to call, **close_finger()**.

This is called “dot notation”, and is very common throughout most object-oriented programming languages.

There's a lot more to learn on this topic, but to get started with Arduino programming, this level of basic understanding of object orientation can take you a long way.

Inputs and outputs

Inputs and output are a fundamental feature of the microcontroller. You can connect devices to special pins on your Arduino, and read or change the state of these pins through special instructions in your sketch.

There are two kinds of input and output pins on an Arduino: digital and an analog.

Let's have a look at them going forwards.

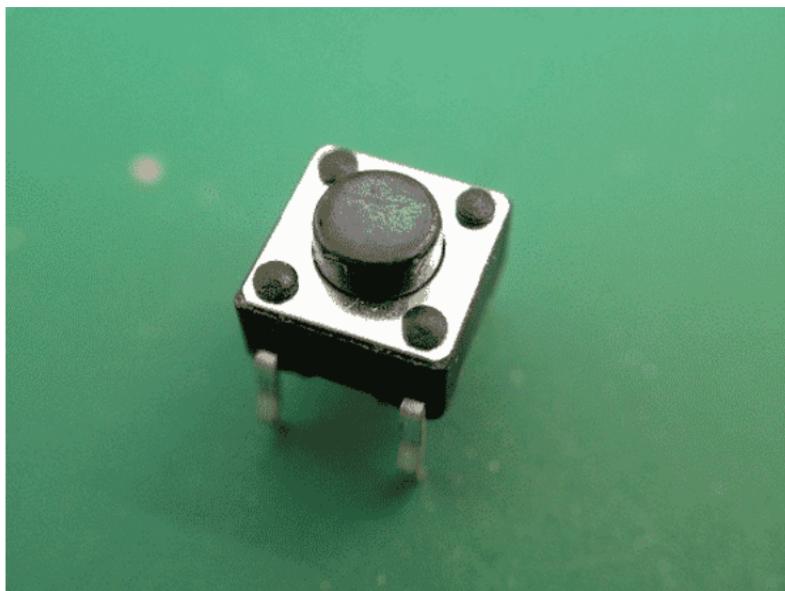
Digital pins

Digital pins are useful for reading the state of devices like buttons and switches or controlling things like relays and transistors or LEDs. These examples have one thing in

common: they only have two possible states.

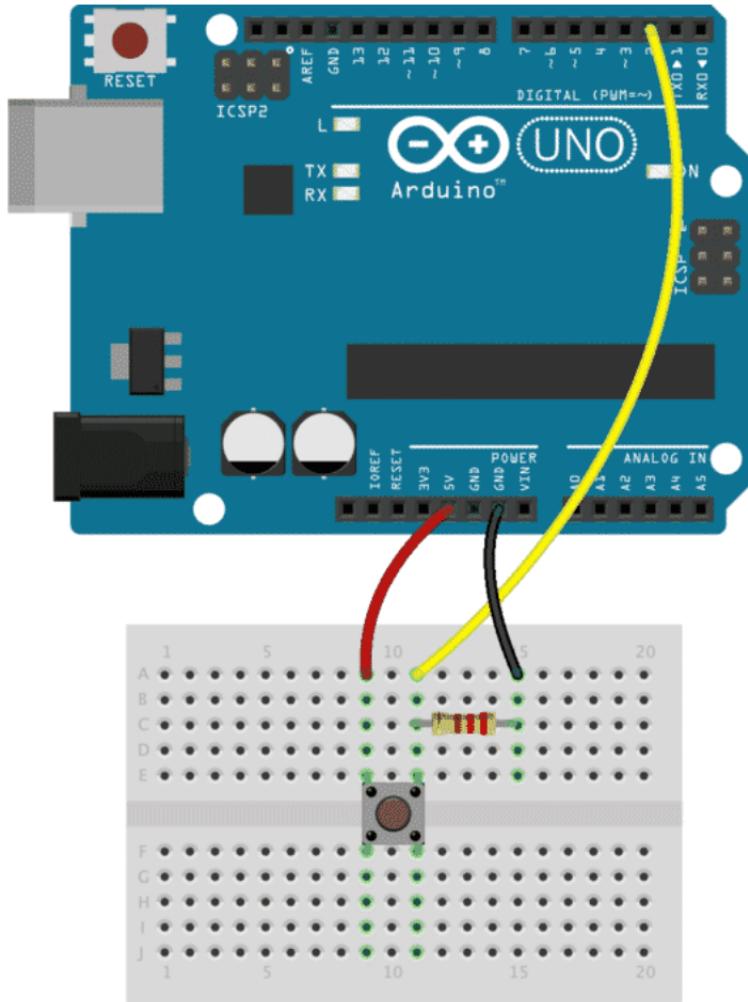
A button can be either pressed on not pressed. A switch can be on or off. A relay can be energized or not.

If in your sketch, you want to know the state of a button, you can connect it to a digital pin. You can wire it up so that when the button is pressed, a 5V voltage is read by the connected digital pin, and that is reported as “high” to your sketch.



A button like this one is a digital device. Connect it to a digital pin.

Let's suppose that you connected a button to a digital pin on your Arduino, as I show in this schematic:



fritzing

A button is connected to digital pin 2. There is also a 10K resistor that conveys a 0V signal to pin 2 when the button is not pressed.

When you press the button, the voltage conveyed by the yellow wire to digital pin 2 is 5V, equivalent to “logical high.” This happens because when the button is pressed, internally, the red wire coming from the 5V source on the Arduino is

connected electrically to the yellow wire that goes to pin 2.

When the button is not pressed, the voltage at pin 2 is 0V, equivalent to “logical low.” This happens because of the resistor in the schematic. When the button is not pressed, the yellow wire is connected to the GND pin on the Arduino, which is at 0V; thus, this level is transmitted to pin 2.

You can read the state of the button in your Arduino sketch like this:

```
int buttonState = 0; void setup() {pinMode(2, INPUT); } void loop(){buttonState = digitalRead(2);if (buttonState == HIGH){ //Do something when the button is pressed} else{//Do something else when the button is not pressed}}
```

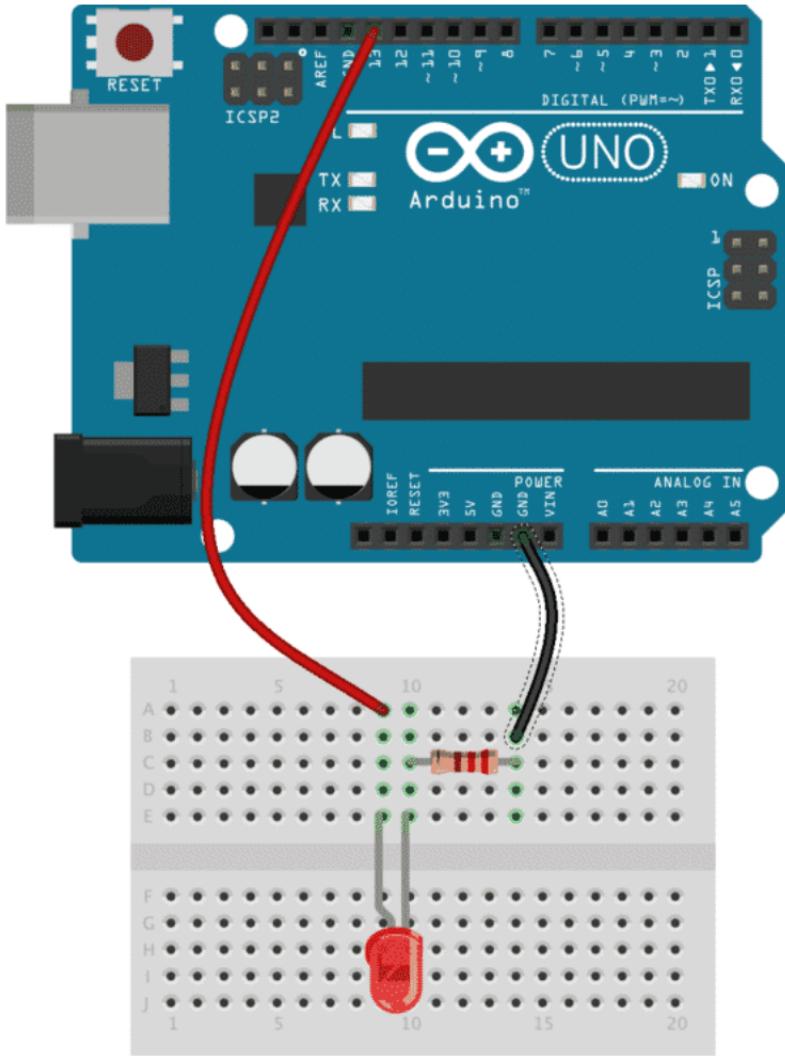
First, create a variable to hold the state of the button.

Then, in the **setup()** method, tell the Arduino that you will be using digital pin 2 as an input.

Finally, in the **loop()**, take a reading from digital pin 2 and store it in the **buttonState** variable.

We can get the Arduino to perform a particular function when the button is in a specific state by using the **if** conditional structure.

What about writing a value to a digital pin? Let’s use an LED as an example. See this schematic:



An LED is connected to digital pin 13. A 220 resistor protects the LED from too much current flowing through it.

In this example, we have a 5mm red LED connected to digital pin 13. We also have a small resistor to prevent burning out the LED (it is a “current limiting resistor”). To turn the LED on and off, we can use a sketch like this:

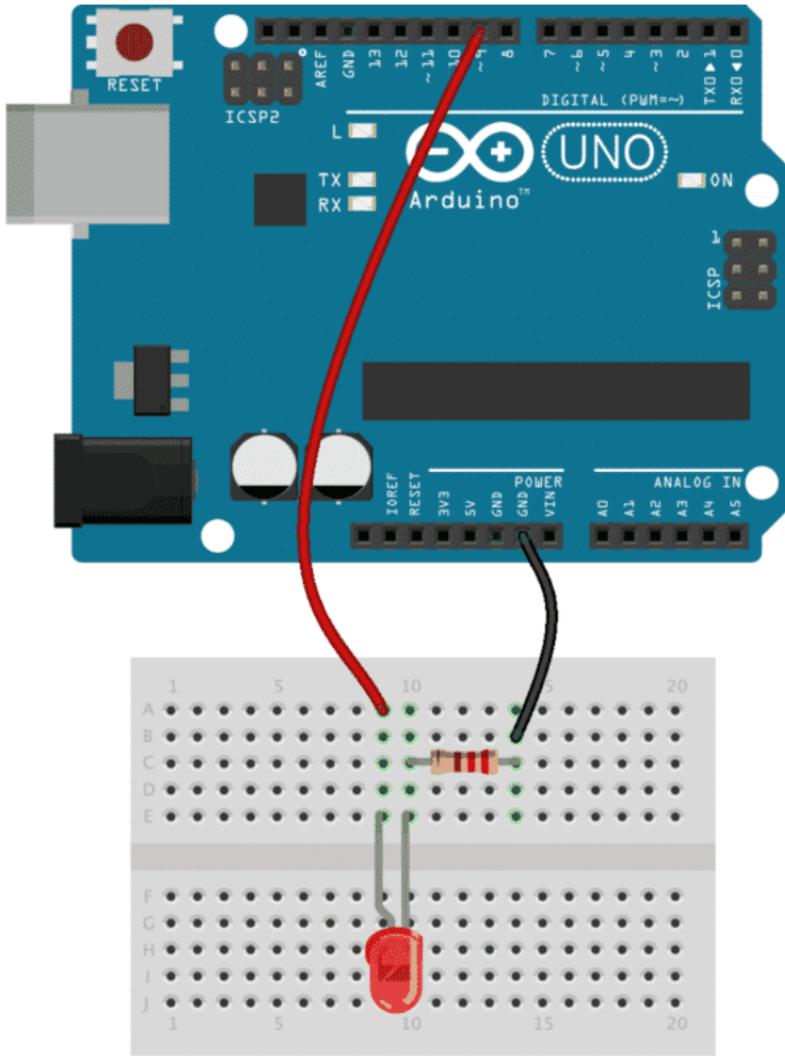
```
void setup() {pinMode(13, OUTPUT);}void loop()
{digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage
level)delay(1000); // wait for a seconddigitalWrite(13, LOW); //
turn the LED off by making the voltage LOWdelay(1000); // 
wait for a second}
```

Just like the button example, first, we must tell the Arduino that we wish to use digital pin 13 as an output. We do this in the **setup()** function with `pinMode(13,OUTPUT)`. In the **loop()** function, we use the **digitalWrite** function to write logical HIGH and LOW to digital pin 13. Each time we change the state, we wait for 1000ms (=1 second). The Arduino has been configured to translate logical HIGH to a 5V signal, and logical LOW to a 0V signal.

Analog pins

Let's move to analog now. Analog signals on microcontrollers is a tricky topic. Most microcontrollers can't generate true analog signals. They tend to be better at "reading" analog signals. The ATMEGA328P, which is used on the Arduino Uno, simulates analog signals using a technique called Pulse Width Modulation. The technique is based on generating a pattern of logical HIGHs and LOWs in a way that generates an analog effect to connected analog devices.

Let's look at an example. We'll take the same LED circuit from the digital pins section and make it behave in an analog way. The only difference in the schematic is that you have to change the wire from digital pin 13 to go to digital pin 9 instead. Here is the new schematic:



An LED is connected to digital pin 9. A 220 resistor protects the LED from too much current flowing through it.

In this example, change the red wire to go to digital pin 9 instead of 13. We do this because we want to make the LED fade on and off via pulse width modulation. Pin 9 has this capability, but pin 13 does not.

We have to switch the controlling pin because we want to simulate an analog signal through the use of Pulse Width Modulation (PWM). Only a few of the pins on an Arduino can do this. One of these pins is 9, which we are using in this example.

Before showing you how to write an analog value to a PWM pin, look at this YouTube video to see what the end result is like.

Here is the sketch to make the LED fade on and off:

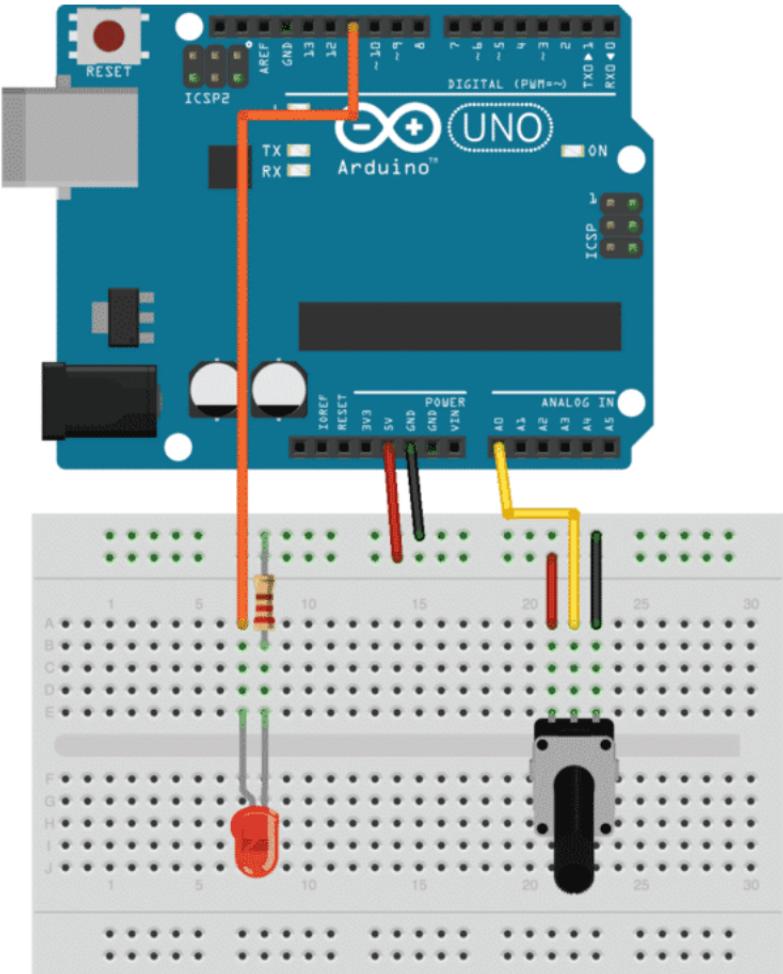
```
void setup() {}void loop() {for (int fadeValue = 0 ; fadeValue <= 255; fadeValue += 5) {analogWrite(9, fadeValue);delay(30);} }
```

In the middle of the **loop()** function, you will find a reference to the **analogWrite** function. This function takes two arguments: the PIN and an 8-bit PWM value.

In the example, the variable **fadeValue** contains a number that changes between 0 and 255 in hops of 5 each time it is **analogWrite** is called because it is inside a **for** loop. When **fadeValue** is at 0, then the **analogWrite** function keeps the output at pin 9 to 0V. When **fadeValue** is at 255, then **analogWrite** keeps the output at pin 9 to 5V. When **fadeValue** is at 127, then **analogWrite** keeps the output at pin 9 at 0V for half of the time and 5V for the other half.

Because the ATMEGA is a fully digital IC, it simulates analog by just switching between digital high and low very quickly. For the LED to be brighter, we give **analogWrite** a larger value, which simply increases the amount of time that the pin stays at logical high versus logical low.

What about reading the state of an analog device? Let's use a potentiometer as an example. This example combines an LED with a potentiometer.



Made with Fritzing.org

The potentiometer in this diagram has its middle pin (signal) connected to analog pin 0.

In this example, when you turn the knob of the potentiometer in one direction, the LED becomes brighter. When you turn it towards the other direction, it becomes fainter.

We want to make the LED brighter when we turn the knob of the potentiometer towards one direction and fainter when we turn it towards the other. To make this happen, we will both

get an analog reading of the state of the potentiometer, and produce PWM output for the LED.

In this video, you can see how the circuit works:

Here is the sketch:

```
void setup() {pinMode(9, OUTPUT);}void loop() {int potValue = analogRead(A0);int brightness = map(potValue,0,1023,0,255);analogWrite(9,brightness);}
```

In the setup function, we set pin 9 to output because this is where we have connected the LED. Pins are inputs by default, so we don't have to set analog pin 0 to be an input explicitly.

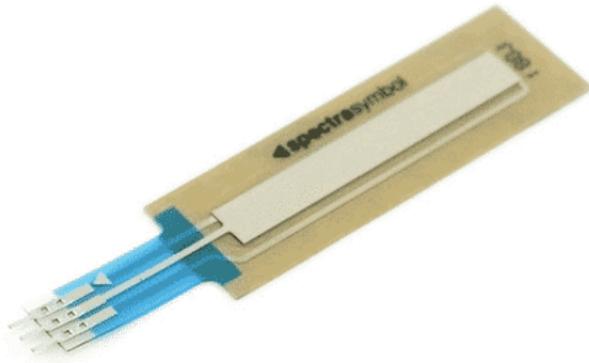
In the loop function, we get a reading from analog pin 0 (its name is "A0") and store it in a local integer variable, **potValue**. The function **analogRead** returns an integer with a range from 0 to 1024. Remember from the earlier example that the PWM function can only deal with the value from 0 to 255. This means that the value we store in **potValue** will not work with **analogWrite**.

To deal with this, we can use the Arduino "map" function. It takes a number that lies within a particular range and returns a number within a new range. So in the second line of the loop function, we create a new local integer variable, **brightness**. We use the map function to take the number stored in **potValue** (which ranges from 0 to 1023) and output an equivalent number that ranges from 0 to 255.

Notice that the parameters of the map function match the range of **potValue** and **brightness**? The conversion calculation is done for you, easy!

Analog read and write are easy once you understand the implications of the available resolution and Pulse Width Modulation. With what you already know, you will be able to work with a multitude of devices using the circuits from the examples in this section.

For example, if you would like to use a membrane potentiometer link this one:



A membrane potentiometer. Electrically it works like a normal rotary potentiometer.

Just remove the rotary potentiometer from the example circuit and replace it with the membrane potentiometer. You will be able to control the brightness of the LED by sliding your finger up and down the membrane.

That wraps up this introductory course on the Arduino!

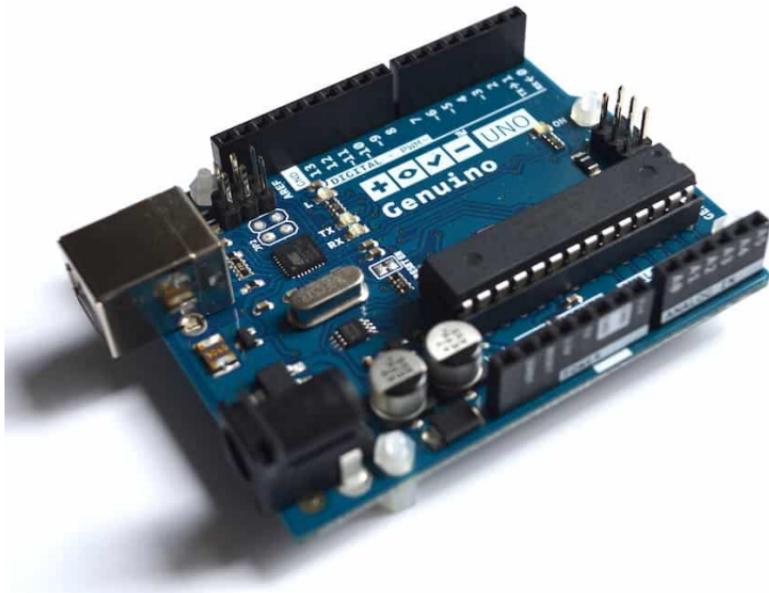
I hope you enjoyed these articles and that you learned something new.

Ready for some serious Arduino learning?

Start right now with Arduino Step by Step Getting Started

This is our most popular Arduino course, packed with high-quality video, mini-projects, and everything you need to learn

Arduino from the ground up.



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

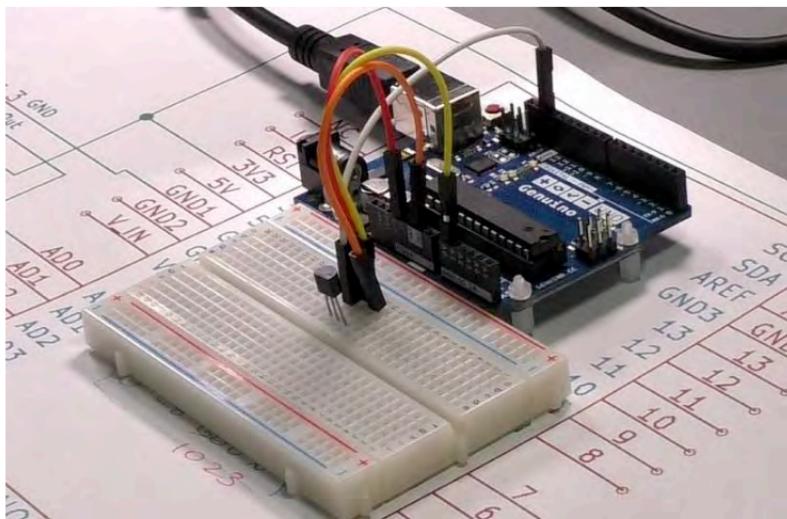
This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

Introduction to Arduino sensors

Arduino Sensors & Actuators guide series

Introduction to Sensors & Actuators

In the tutorials in this series, you will learn how to use an LED and a button, and then go straight into learning how to use several popular and very useful sensors.



So, you are eager to experiment with your Arduino and some of the most commonly used peripherals?

In the tutorials in this series, you will learn how to use an LED and a button, and then go straight into learning how to use several popular and very useful sensors.

By the time you complete these 11 experiments, you'll have a good beginner-level understanding of Arduino programming

and components.

Before you dive right in, please read the following so you can get a good understanding of sensors.

Sensors are the eyes and ears of machines: they provide environmental data. There are all sorts of sensors, some more exotic than others. Here's a shortlist from [Wikipedia](#):

- Light
- Motion
- Temperature
- Magnetic fields
- Gravity
- Humidity and moisture
- Vibration
- Pressure
- Electrical fields
- Sound
- Stretch and stress



Clever gadgets combine multiple sensors in order to capture a more complete snapshot of their environment.

This is similar to our human perception of the environment that is based on multiple senses, like sight and hearing.

Each sensor that is attached to a machine requires processing power. The more sensors attached, the greater the processing requirements on the machine. In the Arduino Uno, the ATMega328 micro-controller is a simple computer running at a clock speed of 16MHz (mega-hertz). This means that this Arduino can process roughly 16 million instructions every second. This processing resource has to be shared between all the things that your Arduino is supposed to do, like reading values from its sensors, doing calculations, updating a screen or other outputs, communicating with other devices, and interacting with the user.

The Arduino is fast, but it has a limit, and your design must

take that into consideration.

What's next

In the lectures in this section, we will play with the following components:

- An LED and how it works.
- A button, and how it works.
- A photo-resistor for measuring light.
- Combined temperature and humidity.
- Infrared line sensor.
- Barometric sensor for measuring air pressure.
- Ultrasonic sensor for measuring distance to other objects.
- Tilting, so you know if your gadget has fallen over.
- Orientation.

How sensors work

Simple sensors, like the photo-resistor for measuring light, work by measuring the voltage they provide to one of the analog sockets in the Arduino. You can do this by using the `analogRead` function. Other sensors are a bit more involved, and they require special software libraries to work with the hardware. More often than not, however, these libraries are very easy to learn and they provide useful extra features at no additional cost.

Blinking LED

Arduino Sensors & Actuators guide series

How to make an LED blink on and off

A Light Emitting Diode (LED) is a special kind of a diode that can emit light.



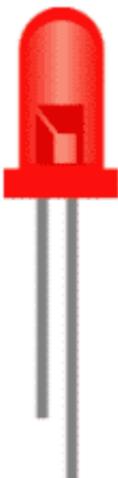
A Light Emitting Diode is a special kind of a diode in that it can emit light. Although all diodes do emit light, in most cases this light is not bright enough so we can't see it. LEDs are specially constructed to allow the light produced to escape outwards so we can see it instead of just being absorbed by the semiconductor.

A diode is a polarised device. In a polarised device, there is a

correct way and an incorrect way of connecting it to your circuit. Connecting a polarised device to your circuit incorrectly will, at least, result in your device not working. In some cases the device can burn out.



LEDs come in several colours, including white. There are even LEDs that can generate thousands of colours by combining the three primary: red, green, blue.



This is what an LED device looks like. Notice that there is a short and a long "leg". The short leg is called "the cathode", often noted with a "k" and should be connected to the negative ("−") voltage. The longer leg is called "the anode", often noted with an "a" and should be connected to the positive ("+") voltage. Other devices of note that are polarised and use similar or same terminology are transistors and certain types of capacitors.

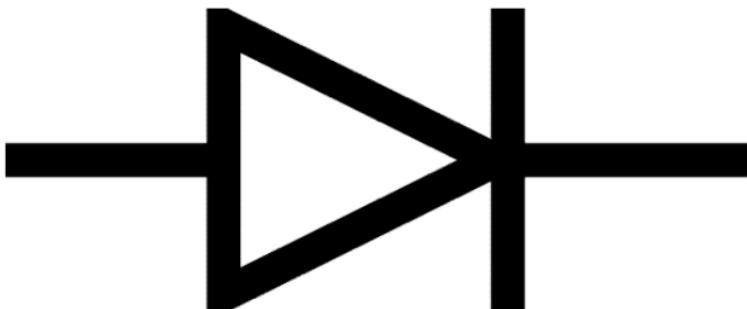
Symbolically, i.e. in diagrams depicting electronic circuits, an LED is depicted like this:



The schematic symbol for an LED

The basic characteristic of a diode is that it is a semiconducting device that allows the flow of electricity (electrons) only towards one direction. Think of it as the equivalent of a plumbing valve that allows water to flow only in one direction. Therefore, diodes are used in situations where we want to restrict the directionality of electricity. Diodes are used extensively in applications like the conversion of current from alternating to direct, in radio transmitters for the modulation of signals, and many others.

The symbol of a diode is almost the same as the one for an LED. The only difference is the presence of three little arrows which show that light is emitted from an LED:



The schematic symbol for a regular LED

Simulation

I have recorded a short video to show you how to run this experiment inside a simulator. This is the best way to experiment with the Arduino if you don't have the real thing. Be sure to read the complete tutorial to understand the details of the circuit and the sketch.

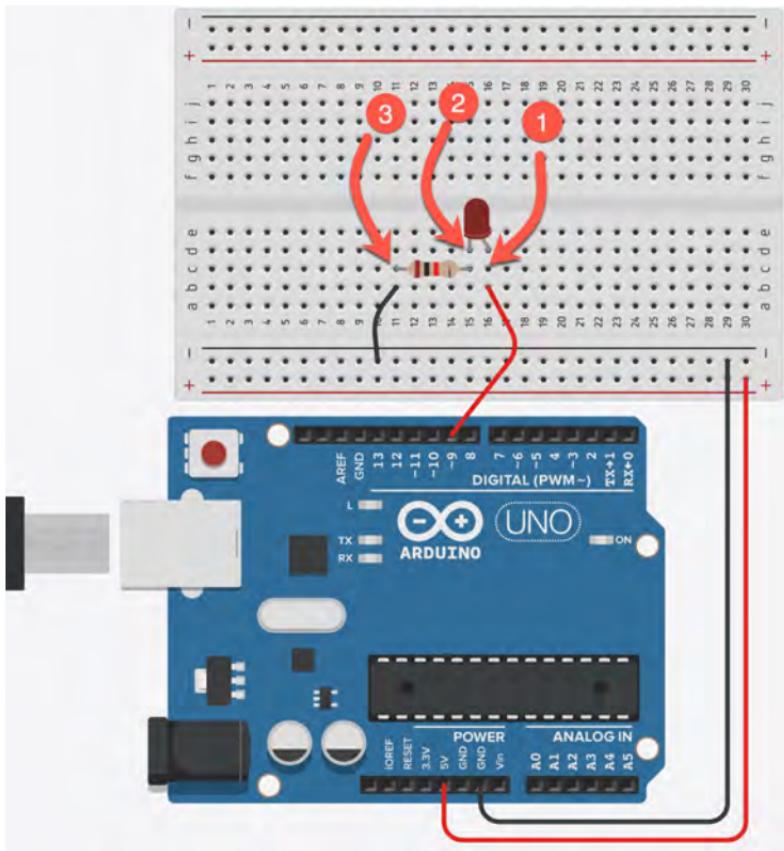
But if you are in a hurry, watch this video.

Let's experiment with an LED

With the background and theory behind us, let's implement our first Arduino circuit. The aim is to become familiar with plugging components into the breadboard, uploading and running a sketch.

In our first sketch we will simply make an LED blink on and off.

Here's a diagram of the circuit you need to build now.



The wiring diagram for the “blinking LED” experiment.

Follow these steps to wire this circuit:

1. Take an LED, and notice that it has two pins, with one longer than the other. The longer pin is the anode, and the short pin is the cathode. We always connect the cathode towards the GND pin of the Arduino.
2. Take a resistor (it can be any value between 220 Ohm and 500 Ohm, and this experiment will work well), and connect one pin in one of the sockets of column “2” in the breadboard. It doesn’t matter which pin it is since the resistor, unlike the LED, can work the same either way you connect it. Connect the other pin of the resistor in a socket in an empty column (make this column “3”).

Connecting a resistor in series with an LED is important. The LED has a very small resistance on its own. **If you connect it to your Arduino without a resistor, it will result to a large current flowing through it. This can damage the LED, but worse, it can damage your Arduino.**

This is described mathematically by Ohm’s Law, which states that **R=V/I**, where R is the resistance of a device, V is the voltage at its two connectors, and I the current that flows through it. If you solve this equation for I, you get that **I=V/R**. For an LED, R is almost zero, so no matter what the V is, the I will be a very large number.

Want to learn

electronics?



Our course “Basic Electronics for Arduino Makers” is designed specifically for people that use the Arduino as a learning and creativity tool.

This course will teach you the basics of electronics so that you know what things like current limiting resistors, voltage dividers, power supplies, transistor switches, Ohm’s Law, and lots more, actually are.

[Learn More](#)

The LED is a “POLARISED device”. What does this mean?

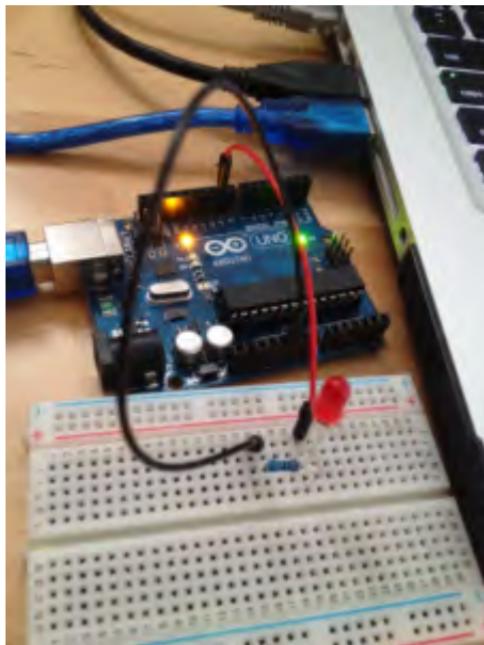
In electronics, there are several devices, like an LED, that are “polarised”. This means that you have to be careful how you connect them to a circuit.

Polarised devices have a positive and a negative pin. The positive pin is called “anode”, and the negative is called “cathode”.

Examples of polarised devices, apart from the LED, are electrolytic capacitors, batteries, diodes, and operational amplifiers.

In the circuit of this experiment, the tricky part has to do with connecting the LED in accordance with its polarity. The “rule of thumb” is to remember that the anode (the positive pin) is longer than the cathode (the negative pin), just like any positive number is bigger than any negative number. Always connect the cathode towards the GND pin of your Arduino.

To finish up with the wiring, plug the Arduino USB cable into an available USB port in your computer. What you should have now is something like this:



Connect your Arduino to your computer via the USB cable.

The program, a.k.a. “sketch”

The LED isn’t doing anything at the moment. To get it to light up and blink, we need to upload a sketch to the Arduino with the appropriate instructions.

Here’s the program we’ll use. I’ll explain how it works. You can

either type it into a new Arduino IDE editor window, or load it by selecting File > Examples > 01. Basics > Blink from the Arduino IDE menu. If you do this, remember to change the number “13” to “9” for variable “led”. I also made it available as a text file download from the materials tab.

[You can get this sketch from Github.](#)

```
// give a name to the pin to which the LED is connected:int led  
= 9;// the setup routine runs once when you press reset:void  
setup() { // initialize the digital pin as an output. pinMode(led,  
OUTPUT);}// the loop routine runs over and over again  
forever:void loop() { digitalWrite(led, HIGH); // turn the LED on  
delay(1000); // wait for a second digitalWrite(led, LOW); // turn  
the LED off delay(1000); // wait for a second}
```

Because this is your first ever Arduino program, I will explain a few things before continuing:

Need a programming refresher?

If you wish, you can also revisit the programming tutorials [Part 1](#) and [Part 2](#) where I go into more detail on the basics of programming the Arduino.

Comments

Any text following “//” or in-between “/*” and “*/” is a comment, and the Arduino will ignore it.

People use these symbols to type comments, like in this example.

Functions

An Arduino program can be broken down in parts by using functions. Functions make it easy to create little programs within a large program, and to call each of these little programs by name.

In this example, we used two functions with names *setup()* and *loop()*.

These are special functions that the Arduino will call itself. When the Arduino starts, it will first call the *setup()* method and execute any commands it finds inside. Then, it will call *loop()* again and again until you turn off the power, every time executing whatever commands it finds inside. You can create your own functions and name them whatever you like, as long as you don't use a reserved name (like "loop" or "setup").

Function names can't have white spaces or other "special" characters inside them.

A function may or may not return a value when it finishes its execution. Notice that loop is declared as void *loop()*? The *void* means that loop does not return anything when it finishes its execution. Same thing happens with *setup()*.

Local and global variables

If you want to store and retrieve values in your sketch, you need to use variables. Variables can store numbers, text, booleans (true/false values) and other data types. In the very beginning of this example, we have this statement:

```
int led = 9;
```

This creates a *global variable* named *led*, and stores the value 9 in it, which is of type *int* (integer).

A global variable is accessible from anywhere in your sketch.

In this example, you can see that inside the *setup()* function, there is a reference to "led", and similarly there is a reference to "led" from inside the *loop()* function.

On the other hand, a local variable is one that is only accessible from within its own context.

If we had declared the *led* variable inside the *setup()* function, then it would only be accessible by other statements inside the *setup()* function and would not be accessible from the *loop()* function.

Arduino functions

Arduino's magic is in the functions that are build-in to the language. These functions make it easy to control many aspects of our hardware.

Notice that in the *setup()* function, there is a call to the function *pinMode*. This function takes in two parameters: first the number of the pin we want to configure, and second the mode that we want to assign to this pin.

In our example, we have *pinMode(led, OUTPUT);*. Remember that we have a global variable called *led* in which we stored the number "9". So we can re-write the *pinMode* instruction as *pinMode(9, OUTPUT);*.

This means that we configure pin 9, which is a digital pin with possible states HIGH (5V) and LOW (GND) to be an output. Being an output, this pin can be used to send a value to a connected device, in our case the LED.

With the *setup()* function complete, the Arduino then starts calling the *loop()* function. The first thing that happens there is calling the *digitalWrite* function:

```
digitalWrite(led, HIGH);
```

With *digitalWrite*, we assign a new state to a pin. We can rewrite this statement as *digitalWrite(9, HIGH);*, and, as you can probably guess, we are changing the state of pin 9 to HIGH, which is 5V. As soon as this happens, your LED will light up!

We want to keep this LED lit for a little while, so we use the instruction *delay* to keep things as they are. *delay* accepts one

parameter, that is the number of milliseconds to wait for.

So in our case:

```
delay(1000);
```

means: "wait for 1000 milliseconds", which is 1 second.

Then we call *digitalWrite* again, but this time we change the state of pin 9 to LOW, which is 0 Volts.

```
digitalWrite(9, LOW);
```

We wait at this state for another second, then the loop starts all over again.

So there you have it, your first Arduino circuit, and a blinking LED!

In the next tutorial, we will make the same LED, using the exact same circuit, fade on and off, giving us a much nicer visual effect to look at.

Ready for some serious Arduino learning?

Start right now with Arduino Step by Step Getting Started

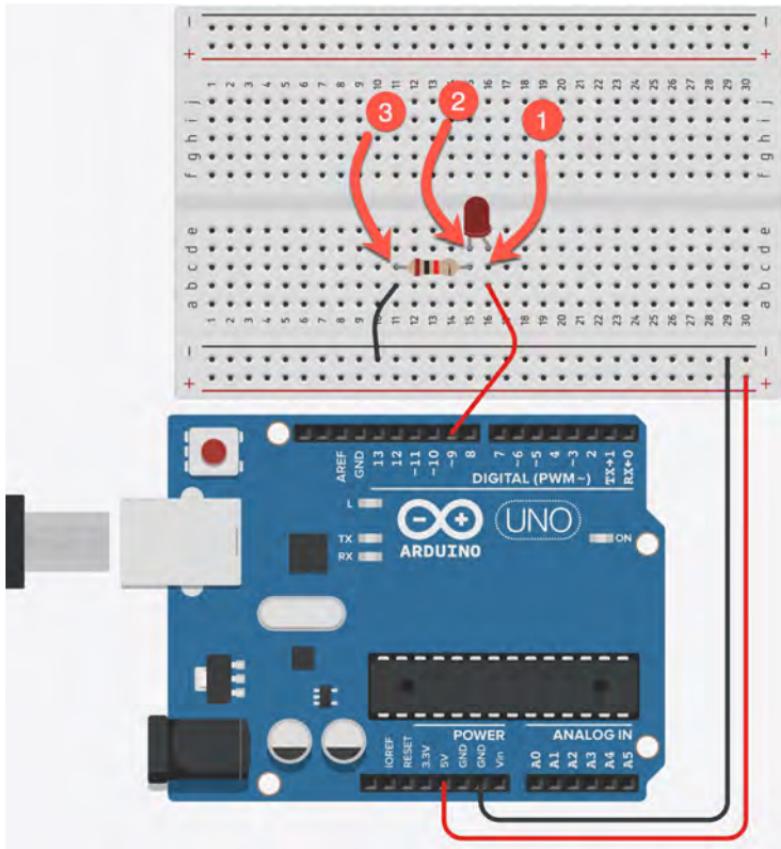
This is our most popular Arduino course, packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up.

Fading LED

Arduino Sensors & Actuators guide series

How to make an LED fade on and off

In this article, you will learn how to make an LED fade on and off instead of simply blinking.



In the previous tutorial, you learned how to setup a simple circuit in which an LED blinks on and off. The Arduino sketch that drove the circuit simply wrote a HIGH or LOW value to the digital output pin 9, and the LED was turn on or off accordingly.

In this article, you will learn how to make the LED not blink but fade on and off. You will keep the exact same circuit, and only change the sketch to make this happen.

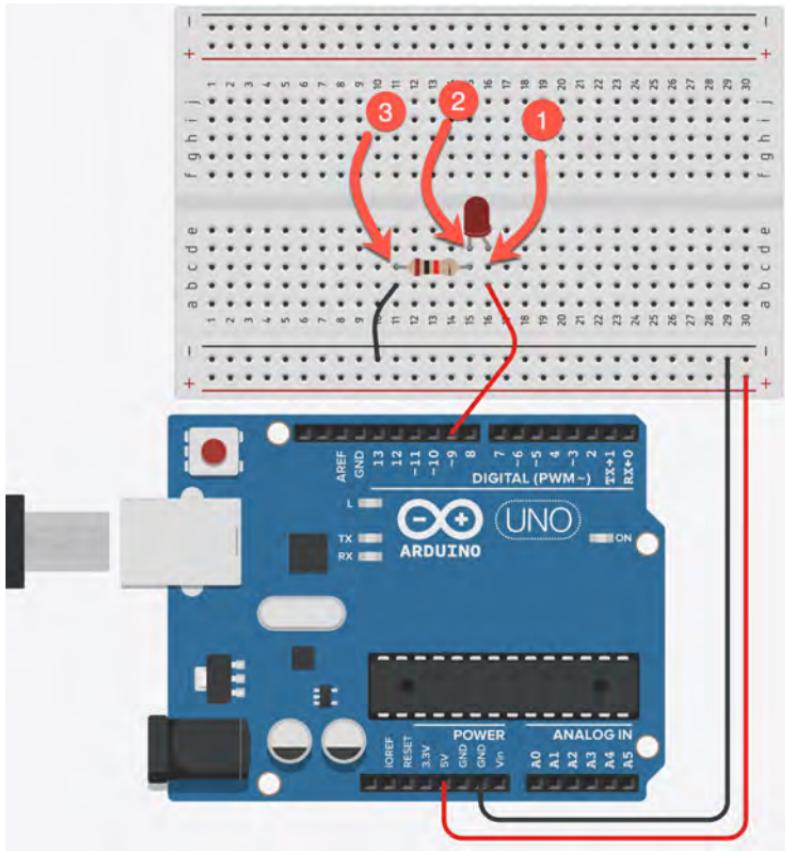
Simulation

I have recorded a short video to show you how to run this experiment inside a simulator. This is the best way to experiment with the Arduino if you don't have the real thing. Be sure to read the complete tutorial to understand the details of the circuit and the sketch.

If you are in a hurry, watch this video first.

The wiring diagram

Here is the wiring diagram, in case you need it:



The wiring diagram for the “blinking LED” experiment.

The sketch

And, here's the new sketch (slightly modified from the sample in the Arduino IDE which you can load by going to File > Examples > 01.Basics > Fade) (or [you can get this sketch from Github](#)):

```
int led = 9; // the pin that the LED is attached to  
int brightness = 0; // the bigger this number, the brighter the LED is  
int fadeAmount = 5; // the bigger this number, the faster the LED will fade on or off  
void setup() {  
  pinMode(led, OUTPUT); // declare pin
```

```
9 to be an output:}// the loop routine runs over and over again
forever:void loop() { analogWrite(led, brightness); // set the
brightness of pin 9: brightness = brightness + fadeAmount; //
change the brightness for next time through the loop: //
reverse the direction of the fading at the ends // of the fade: if
(brightness == 0 || brightness == 255) { fadeAmount = -
fadeAmount ; } delay(10); // wait for 10 milliseconds to see the
dimming effect}
```

The most important difference between this sketch and the blinking LED sketch, is that we now use *analogWrite* instead of *digitalWrite*.

While *digitalWrite* can only output a HIGH or LOW value, *analogWrite* allows us to output values between 0 and 255.

The *analogWrite* function uses a technique called “Pulse Width Modulation”, or PWM for the abbreviated short. A the function takes in one argument, a number from 0 to 255. The Arduino will then convert this value into a square waveform. The square waveform has a high of 5V and a low of 0V, but depending on the PWM value we set in *analogWrite*, the duration of HIGH varies.

When we set PWM value to 255, the square wave is at HIGH permanently and the connected LED is at its brightest setting.

When we set PWM to 0, the square wave is at 0V permenently, and the LED is switched off.

In-between values, like 120, result to the square wave being HIGH for around half of the period, and LOW for the rest of the period. That way, the LED is around half as bright as it can be.

The amount of time of a period that a PWM signal is at HIGH is known as the “duty cycle”.

Using PWM, a simple device like the Arduino can simulate an analog output signal, which as a real analog effect on a device like an LED or a motor. PWM gives us a simple way to set an

LED to 1/3 (or any other fraction) of its full potential brightness.

Similarly, we can control the speed of a motor or the loudness of a speaker.

In this video, I show what a PWM signal that is produced by an Arduino looks like in the oscilloscope. This signal causes the LED to fade on/off.

WANT TO LEARN HOW TO USE AN OSCILLOSCOPE?



Do you think that the oscilloscope is too complicated, too expensive and has no place on your workbench? You're wrong! With this course, you will learn how to use the oscilloscope and take your understanding of electronics to the next level.

In the *loop()* function, we first set the brightness of the LED,

using the *analogWrite()* function, by selecting the pin to which the LED is connected, and the ‘brightness’.

Brightness is a global variable of type integer that is initialised to be 0 when the program starts. So, the first time that the program runs in the Arduino, this instruction will look like this:

```
analogWrite(9, 0);
```

In the next instruction, after the comment, we calculate a new value for the brightness. The new brightness is equal to the old brightness plus the *fadeAmount*, the value stored in another global variable that we set to be 5 in the very start of the program. So, the first time the program runs, this instruction will look like this:

```
brightness = 0 + 5;
```

Therefore, brightness will now become 5.

Next, we use a control structure to determine if we have reached a limit for the brightness, either the lower limit (0) or the upper limit (255). If we have, then we switch the sign of the *fadeAmount* variable. The effect is that if the LED was becoming brighter because the *fadeAmount* was positive, then once it reaches its brightest setting (when brightness equals 255), then *fadeAmount* will be changed its negative (-5) and brightness will start moving towards zero.

In the statement:

```
if (brightness == 0 || brightness == 255)
```

...the part within the parentheses is called a “condition”. The “`==`” tests for equality. You could test for “greater” with “`>`” or “less” with “`<`”, as well as for a variety of other conditions.

In the same statement, the “`||`” is the boolean operator “**OR**”. You can join two conditions together, and the result will be true if one of them is true. So, in our example, whatever is

between the curly brackets will be executed if either brightness is zero **OR** brightness is 255.

Ready for some serious Arduino learning?

Start right now with Arduino Step by Step Getting Started

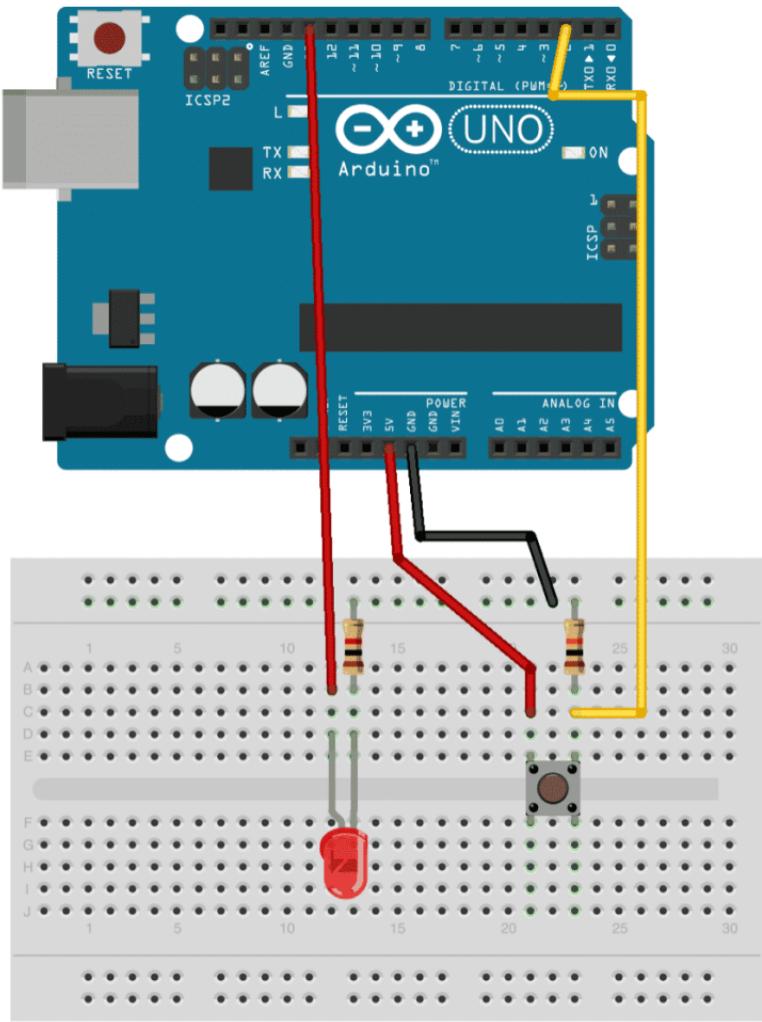
This is our most popular Arduino course, packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up.

Button

Arduino Sensors & Actuators guide series

How to use a momentary button

In this article, I will show you how to use a momentary button with the Arduino.



A button is a simple on-off switch. There are many kinds of buttons, distinguished by the mechanism used to close or open a circuit, but essentially all buttons belong to one of two families: those that keep the connection in either an open or a closed state, and those that return to their original (default) state.

In the image below you can see some examples of switches.



Three types of switches: (1) Toggle switch, (2) On/off switch, (3) momentary button

In this image you can see:

1. A toggle switch. It stays open or closed after pushing it.
2. An on/off switch that also remains open or closed.
3. A momentary button that remains closed while pressure is applied to it, then returns to the open position.

A keyboard key or a door bell button are both momentary buttons. Momentary buttons are also known as “biased” because they have a tendency to return to their original position. A light switch, on the other hand, stays at the position it was put in, so it is often called “un-biased”.

Simulation

I have recorded a short video to show you how to run this experiment inside a simulator. This is the best way to experiment with the Arduino if you don't have the real thing. Be sure to read the complete tutorial to understand the details of the circuit and the sketch.

But if you are in a hurry, watch this video.

Experiment

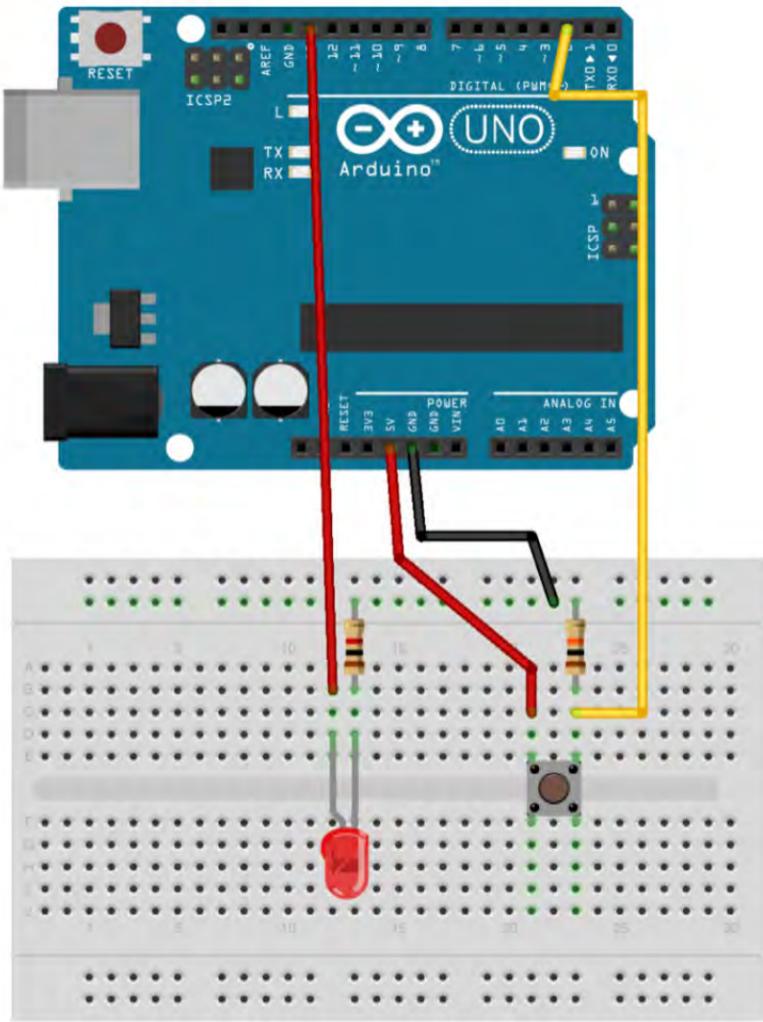
Let's create a simple circuit to demonstrate a button.

We don't really need the Arduino for this. A battery, an LED, a resistor and the button itself would suffice.

But, using the Arduino is simple enough, and it's already got an LED in pin 13 we can use anyway, and no need to worry about a battery pack.

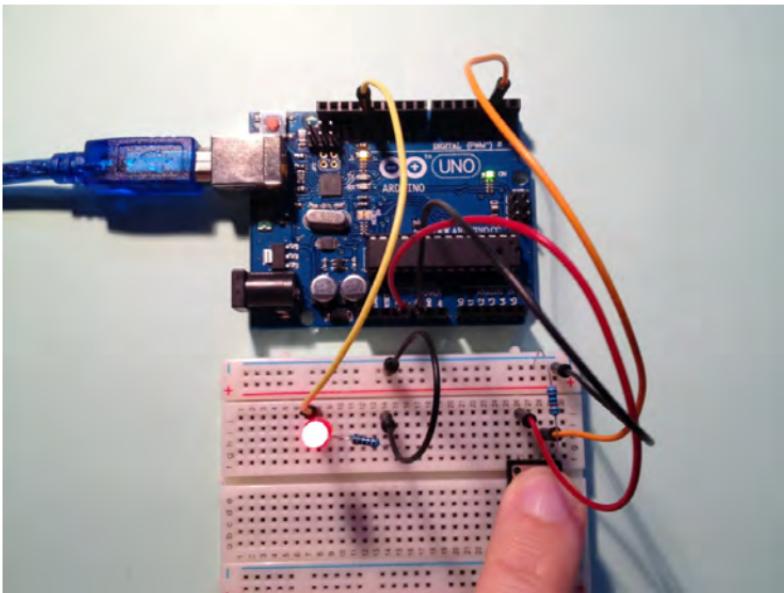
Plus, we can use the monitor to actually see a message when the button is pressed.

So, we'll setup this circuit:



A simple button circuit. When you press the button, the LED turns on.

Once assembled, your circuit will look like this:



The button circuit on a breadboard.

The sketch

The sketch is simply taking a reading from digital pin 2, where one of the pins of the button is connected, and writing a value to digital pin 13, where the LED is connected.

I could have written this script to be even smaller, but I will leave that for you to do as an exercise.

[Here is the sketch on Github.](#)

```
/* Pushbutton sketch a switch connected to pin 2 lights the
LED on pin 13*/const int ledPin = 13; // choose the pin for the
LEDconst int inputPin = 2; // choose the input pin (for a //
pushbutton)void setup() {pinMode(ledPin, OUTPUT); // declare
LED as output pinMode(inputPin, INPUT); // declare pushbutton
as input}void loop(){int val = digitalRead(inputPin); // read
input valueif (val == HIGH){digitalWrite(ledPin,HIGH);}
else{digitalWrite(ledPin,LOW); }}
```

And that's how you can use a momentary button with the Arduino!

Ready for some serious Arduino learning?

Start right now with Arduino Step by Step Getting Started

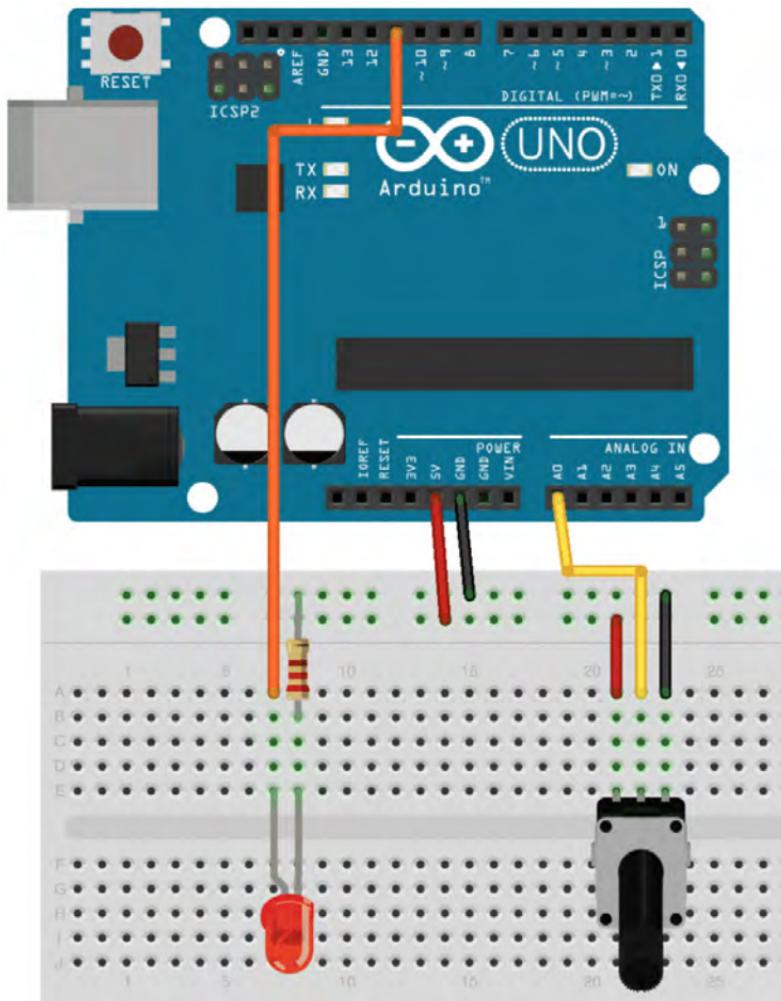
This is our most popular Arduino course, packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up.

Potentiometer

Arduino Sensors & Actuators guide series

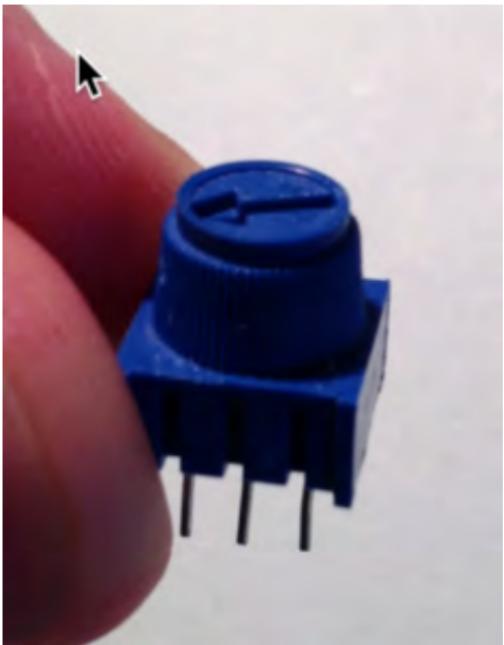
How to use a potentiometer

A potentiometer is a device that allows you to change the value of a resistance by turning a knob. In this tutorial you will learn how to use a potentiometer.



A potentiometer is a device that allows you to change the value of a resistance by turning a knob.

In this tutorial you will learn how to use a potentiometer.



A potentiometer is a very simple sensor. It senses the position of its knob.

Let's put this circuit together, and then we'll discuss how it works. In particular, we'll see what is happening inside the potentiometer.

Simulation

I have recorded a short video to show you how to run this experiment inside a simulator. This is the best way to experiment with the Arduino if you don't have the real thing. Be sure to read the complete tutorial to understand the details of the circuit and the sketch.

But if you are in a hurry, watch this video.

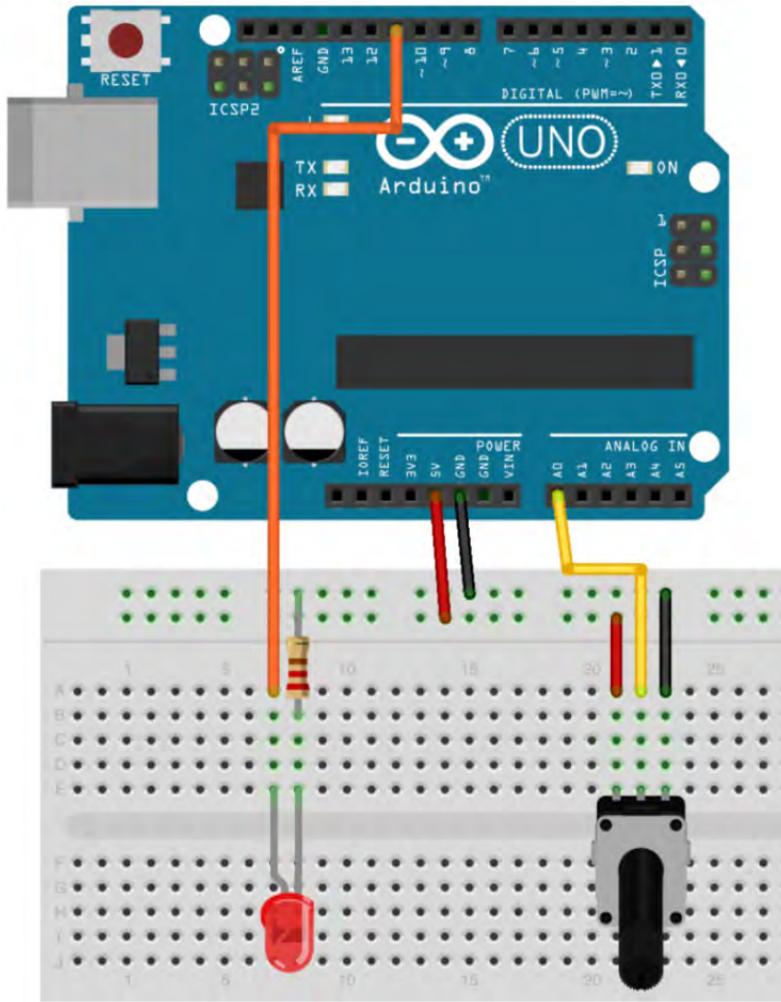
Circuit assembly

Let's assemble the circuit. You will need:

- The Arduino
- A potentiometer
- An LED
- A resistor to protect the LED (I used a 1k resistor, but you can use any value from 220 to 1k)

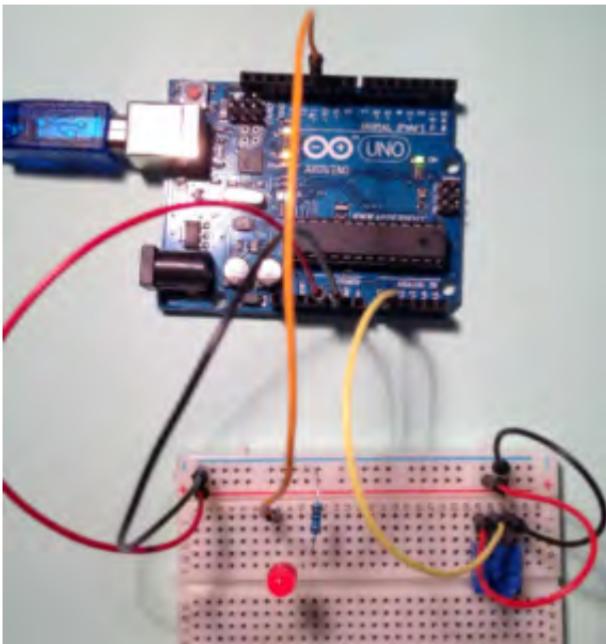
As you turn the knob of the potentiometer, the resistance connected to its output pin changes. This in turn changes the voltage on that pin. This output voltage is read by the Arduino at analog pin 0 (A0).

The Arduino will then use digital pin 11 to drive the LED. Although pin 11 is digital, we are using its Pulse Width Modulation (PWD) feature, like we did back in Lecture 5. We just glossed over this feature back then: we used the analogWrite instruction which makes use of this feature. Later in this lecture I will explain how PWD works.



This circuit uses a potentiometer to control the brightness of an LED.

Here's a photo of the assembled circuit on my breadboard:



The experiment circuit on my breadboard.

Potentiometer, principle of operation

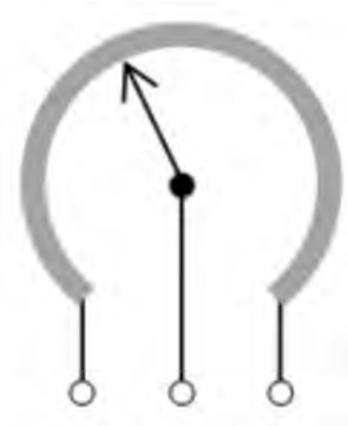
The potentiometer contains a voltage divider. A voltage divider is a simple arrangement of two resistors that can reduce an input voltage to a smaller voltage that depends on the resistors that make up the circuit.

In a potentiometer, we still have these two resistors inside the package, but at least one of them is variable,. This means the resistance of the resistor inside the potentiometer changes as we turn the knob.

In the diagram on the right, the pin in the middle is connected to a dial that is made of conductive material, like copper. They gray-coloured circle represents a resistor, with its two ends connected to V+ and Ground. As the dial comes in contact with different parts of the resistor, it samples a voltage between V+ and ground, and we can read this voltage from

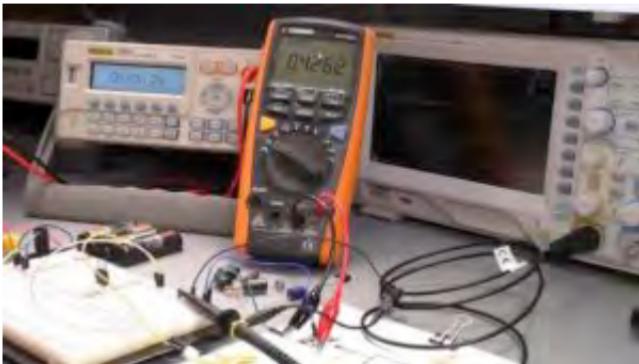
the middle pin.

So there you have it, a potentiometer is nothing more than a variable voltage divider.



A variable resistor, like the one inside a potentiometer.

Want to learn
electronics?



Our course “Basic Electronics for Arduino Makers” is designed specifically for people that use the Arduino as a learning and creativity tool.

This course will teach you the basics of electronics so that you know what things like current limiting resistors, voltage dividers, power supplies, transistor switches, Ohm’s Law, and lots more, actually are.

[Learn More](#)

What is Pulse Width Modulation?

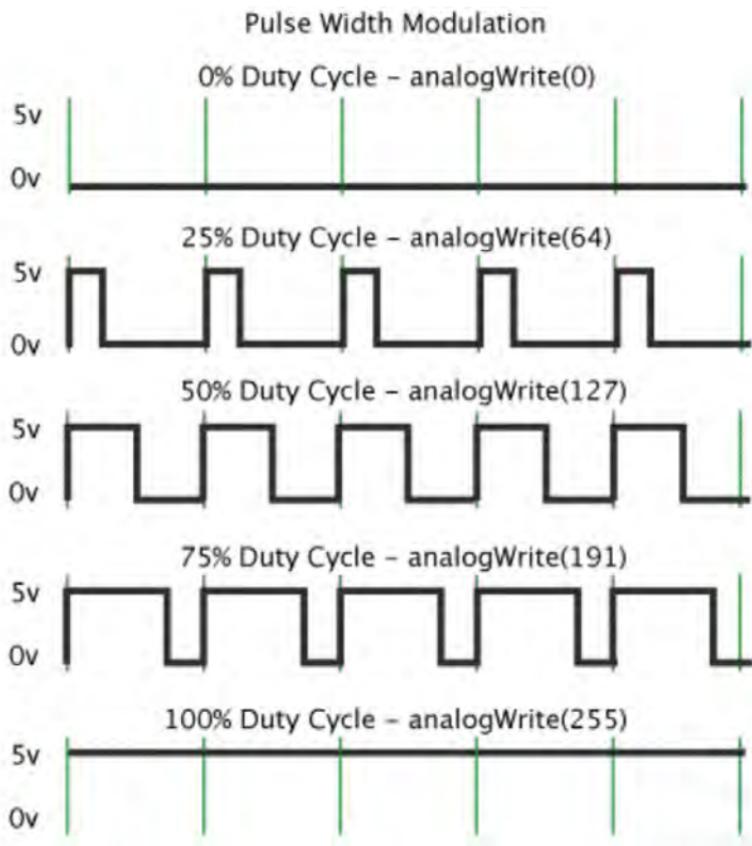
You learned about Pulse Width Modulation [in the tutorial on how to make an LED fade on and off](#). If you haven’t completed that tutorial yet, please do that now because you will need that knowledge going forward. In fact, in this experiment you will use PWM to control an LED, just like you did in the fading LED tutorial. The difference is that in this tutorial, you will control the PWM value through the potentiometer. In the fading LED tutorial, the Arduino controlled the PWM value inside a loop in the sketch.

But since we are on the topic of PWM, let’s have another look at it.

Arduino’s digital pins can output HIGH and LOW, as we have already seen. Some of these digital pins, however, are special:

although they can still only output HIGH and LOW, their output can be modulated in a way that this output can behave as if it was analog.

Have a look at the diagram below, taken from the Arduino web site.



PWM and duty cycles (original: arduino.cc)

A “normal” digital pin outputs 0V or 5V, like in the top and bottom waveform examples. But, in a pin that supports PWM, we can output 5V for only part of the period. In the second waveform, for example, we output 5V for 25% of the period, and 0V for the rest. In the third example, its 50% and 50% respectively.

On the LED, this kind of modulation on the output signal has an effect similar to using an analog pin and outputting voltage in the range of 0V to 5V. In effect, we simulate analog output using a digital pin!

The sketch

Here's the sketch that drives this circuit ([here is the sketch on Github](#)):

```
int potentiometerPin = 0; int ledPin = 11; int potentiometerVal = 0; void setup() { Serial.begin(9600); // setup serial} void loop() { potentiometerVal = analogRead(potentiometerPin); // use the map function because PWM pins can only accept //values from 0 to 255. Analog pins can output values from //0 to 1023. With the map function, the range 0-1023 is //converted to appropriate values from 0 to 255. int mappedVal = map(potentiometerVal,0,1023,0,255); Serial.print(potentiometerVal); Serial.print(" - "); Serial.println(mappedVal); analogWrite(ledPin,mappedVal); delay(10);}
```

There is nothing new here. We have seen the “map” function in an earlier lecture, so you know that this function is used when we want to make a range of values fit within another range of values. In this case, we read values in the range of 0 to 1023 from analog pin 0, and we want to make this fit in the range 0 to 255, which is what the pulse width modulation-capable pin 11 can output.

Other than that, we simply use the “analogRead” function to read a value from analog pin 0, and the “analogWrite” function to create a PWM pulse on digital pin 11.

Ready for some serious Arduino learning?

Start right now with Arduino Step by Step Getting Started

This is our most popular Arduino course, packed with high-

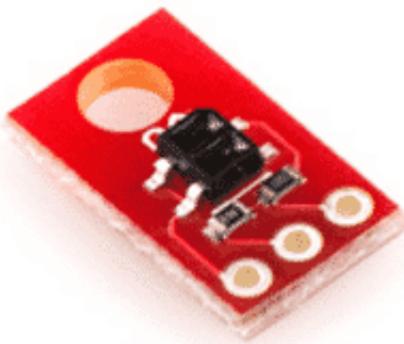
quality video, mini-projects, and everything you need to learn Arduino from the ground up.

Infrared line sensor

Arduino Sensors & Actuators guide series

Infrared line sensor

An infrared line sensor is a simple device made up of an infrared emitting LED and an infrared sensitive photo-resistor. You could use one of these sensors to build a robot that follows a dark line on the floor, or your own heart rate monitor.



An infrared line sensor is a simple device made up of an infrared emitting LED and an infrared sensitive photo-resistor.

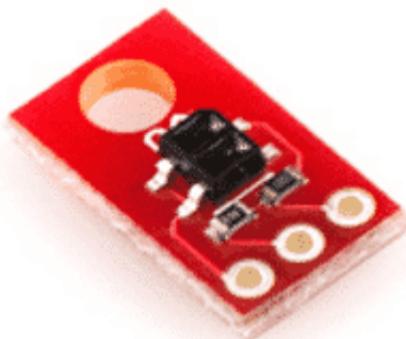
You could use one of these sensors to build a robot that follows a dark line on the floor, or your own heart rate monitor.

The principle of operation is very simple: The transmitter produces infrared light which bounces off a surface and comes back to be captured by the photo-resistor.

The more infrared light is reflected back into the photo-resistor, the higher the output of the sensor gets.

In our experiment we will use a QRE1113 line sensor from

Sparkfun. You can get something like this on eBay for less than \$2.



Assembly

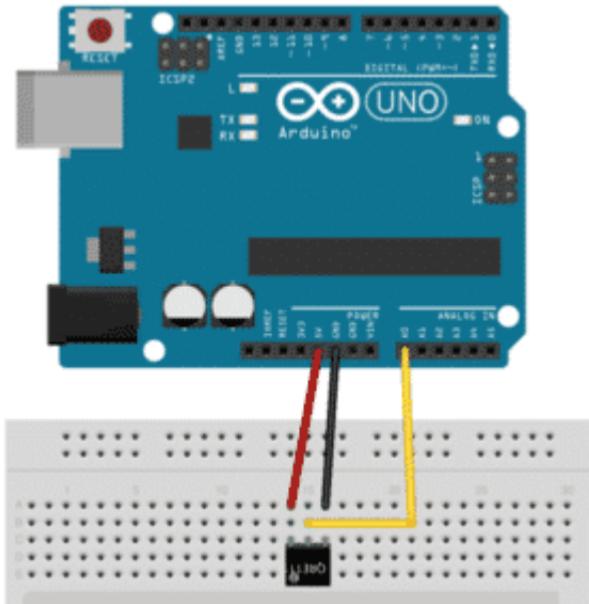
Let's put together this circuit and test out the motion sensor.

We will need:

- The Arduino
- Three jumper wires
- An QRE1113 line sensor or equivalent (like [this](#) ywRobot device).

Here's what we are going to build (below).

For power, you can plug this sensor into either the 3V or 5V sockets on the Arduino.



The completed circuit of the Arduino with an QRE1113 infrared line sensor

Sketch

This one is very simple, just read the analog output at pin A0 and print it to the monitor ([here is the sketch on Github](#)):

```
// Line Sensor Breakout - Analogint out;void
setup(){Serial.begin(9600); // sets the serial port to 9600}void
loop(){out = analogRead(0); // read analog input pin
0Serial.println(out, DEC); // print the value of the
sensordelay(100); // wait 100ms for next reading}
```

First, create the variable that will hold the value from the analog pin, named out.

Inside void setup(), we set the Serial port to 9600 baud rate so that the sensor values can be displayed on the serial monitor while the sketch is running.

Inside the loop() function we read the input from analog pin 0 (A0) and store it in the out variable.

We display the value in the Serial monitor using the Serial.println(out, DEC) command. We use println() (instead of just print()) to display each value in a new line. We use the DEC parameter inside the println() command to display the value as a decimal.

We introduce a 100ms delay to reduce the rate by which the sensor is read.

Ready for some serious Arduino learning?

Start right now with Arduino Step by Step Getting Started

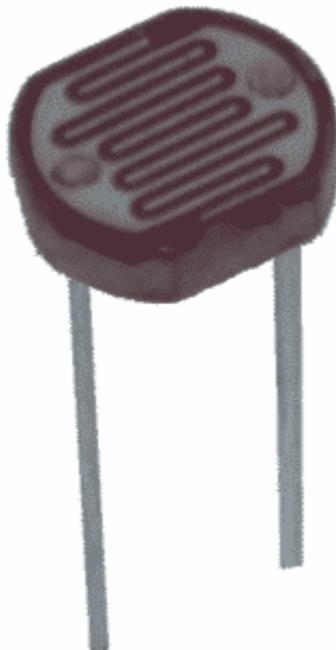
This is our most popular Arduino course, packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up.

Light sensor (analogue)

Arduino Sensors & Actuators guide series

Measuring light

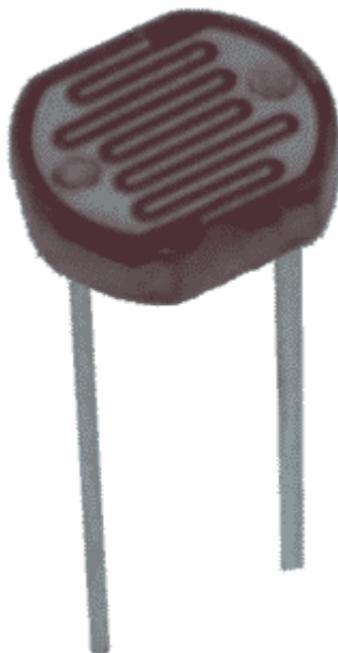
Measuring light with the Arduino is really easy. There are many sensors capable of detecting or measuring light, but the photo-resistor is one of the easiest to use.



Measuring light with the Arduino is really easy. There are many sensors capable of detecting or measuring light, but the photo-resistor is one of the easiest to use. A photo-resistor is simply a resistor in which the resistance changes in accordance to its exposure to light.

The photo-resistor

You can find these devices on eBay for less than \$3 for a pack of ten. Think about what you can do with a device that can detect light. Of course, your gadget will be able to know if its day or night, or if the lights are on. So you could build a gadget that turns on a small light at the entrance of your home when it darkens, so you don't have to walk in the dark.



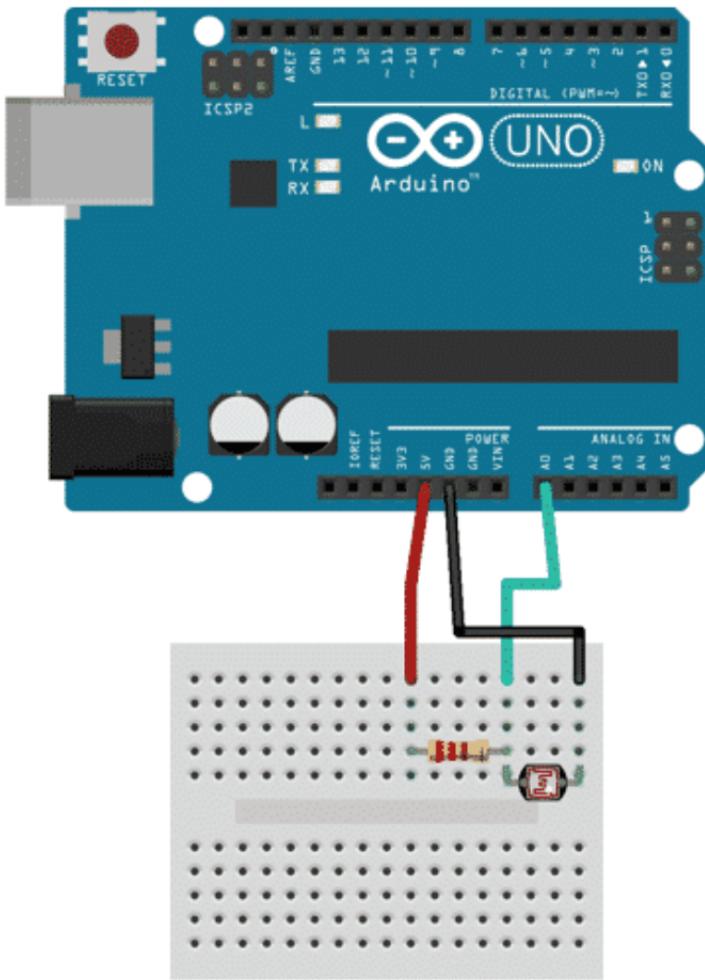
You could also use a photo-sensitive device to allow two gadgets to communicate with light; this is the principle behind the typical television remote control where the remote control and the television communicate using infrared light. You could also build a simple robot that follows a bright or dark line on the floor. Can you think of anything else you could do with a light sensor?

Create a light-sensing circuit

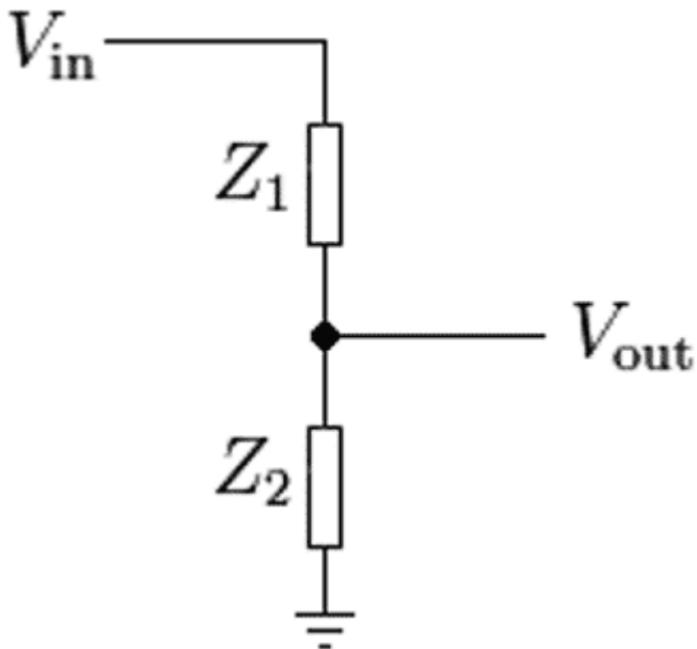
We'll now create a circuit that contains a photo-resistor, and we'll use an Arduino sketch to take light intensity readings from it.

Look at the circuit in the image below, and try to copy it. Here are a few things to be extra careful about:

- Counting from the right, the photo-resistor is connected to a socket in column 1. Its second leg is connected to a socket in column 4.
- We use a 1 kOhm resistor (or close, the exact rating is not important in this exercise) in series with the photo-resistor in order to create a “voltage divider”. More about this in a minute.
- Connect the resistor's second leg to a socket in column 8.
- Connect the black jumper wire to the GND socket on the Arduino, and the red to 5V. Even if you switch these connections, this circuit would still work because we are not using any polarized components.
- Lastly, connect a green jumper wire from a socket in column 4 to A0, which is the Arduino analog port 0.



Notice how you connected the green cable in column 4, where the resistor and the photo-resistor meet? This type of wiring is called a “voltage divider” or “impedance divider”, and its purpose is to create an output voltage that is a fraction of the input voltage to the divider. Look at the diagram (below):



Want to learn
electronics?



Our course “Basic Electronics for Arduino Makers” is designed specifically for people that use the Arduino as a learning and creativity tool.

This course will teach you the basics of electronics so that you know what things like current limiting resistors, voltage dividers, power supplies, transistor switches, Ohm’s Law, and lots more, actually are.

Learn More

Vin is represented by the red jumper wire in our Arduino diagram, the earth symbol is represented by the black jumper wire. The Vout is represented by the green jumper wire. Vout depends on the impedance (resistance) of Z1 and Z2, which in the case of our Arduino circuit are the resistor and the photo-resistor. Since the resistor’s resistance is fixed, and the resistance of the photo-resistor varies depending on the ambient light situation, Vout will also vary. By taking a measurement of Vout, we gain information about the light intensity in our lab.

The higher the measurement in socket A0, the more intense the light is. The closer to zero it gets, the darker our lab is.

That’s enough for now with the hardware. Let’s look at the software.

Sketch

Here’s our program for this exercise ([here is the sketch on Github](#)):

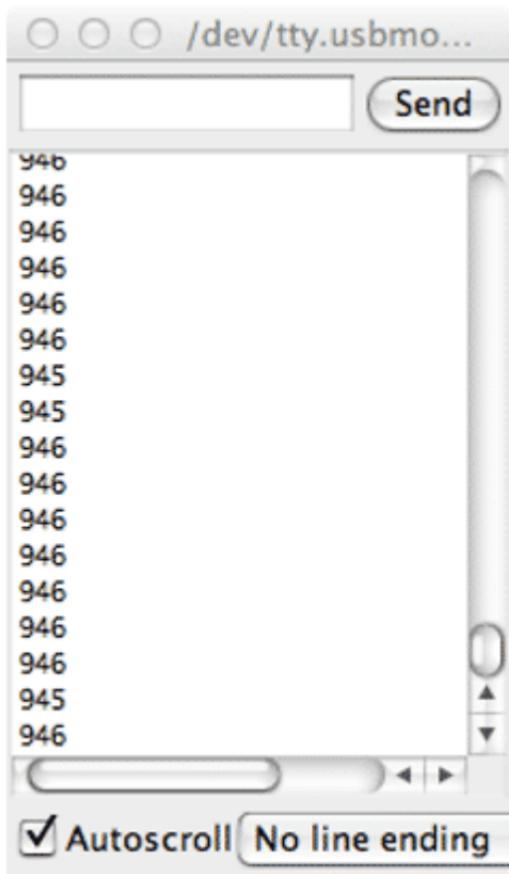
```
// the setup function runs once when you press reset:void  
setup() { // initialize serial communication at 9600 bits per  
second: Serial.begin(9600);}// the loop routine runs over and  
over again forever:void loop() { // read the input on analog pin  
0: int sensorValue = analogRead(A0); // print out the value you  
read: Serial.println(sensorValue); delay(10);}
```

Much of this sketch should be familiar to you now. There's setup(), loop(), and delay(), which we have seen before.

There's a couple of new things here. First, look in the setup() function. There is this statement:

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

This creates a serial connection which the Arduino can use to send text output to our terminal. This way the Arduino can "talk" to us. This terminal can be opened by clicking on Tools > Serial Monitor, and it looks like this:



We will be printing the light intensity value to the serial monitor in a moment.

Next, have a look in the loop() function. In the first actual statement after the comment, we use the analogRead(A0) function to get a reading from socket A0 ("Analog 0") and store it to the local integer variable sensorValue. Easy, right?

We now have a value captured from the photo-resistor's voltage divider circuit, let's print it to the monitor so we can actually see it. Also easy, just do this:

```
Serial.println(sensorValue);
```

This statement, says: "Go to the serial port, print line with content 'sensorValue'". The "In" part of the println() function means that this particular function will create a new line after it prints out the text that is contained within the parentheses. You could use just Serial.print(sensorValue), but then the output would look like this:

```
9469459469459459469459469469459469
```

... not very useful, very hard to read.

When you send this program to the Arduino, wait for it to upload, and then open up the monitor. You will see something like this:

```
946
```

```
946
```

```
946
```

```
946
```

```
946
```

```
946
```

946

946

946

The actual values will vary because of the differences in the components you used in your circuit compared to mine, and of course the lighting conditions in our two labs are probably different. But as long as you see similar values (above 0 and below 1024), then it worked!

Ready for some serious Arduino learning?

Start right now with Arduino Step by Step Getting Started

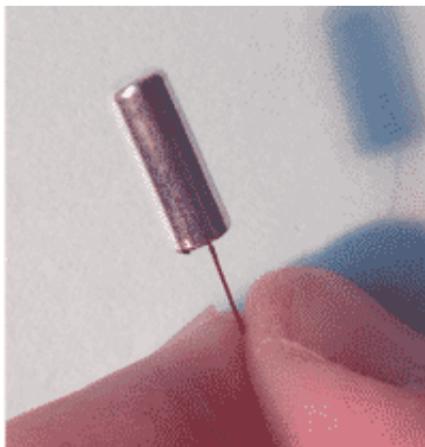
This is our most popular Arduino course, packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up.

Impact sensor

Arduino Sensors & Actuators guide series

Detect tilt and impact

The tilt and impact sensors are simple switches which close a circuit when positioned in a particular way. They are very cheap and come in a variety of shapes. They are usually made of a tiny metallic cylinder with a thin copper wire coming out of one end.



In many cases, knowing the exact force and direction applied to our gadgets is an overkill; just knowing that a bottle has been tipped over is enough to know that the lid should close automatically.

For simple cases like that, a 3-axis accelerometer is an overkill. We could use a simple sensor that can detect the shock of an impact or for being upside down.

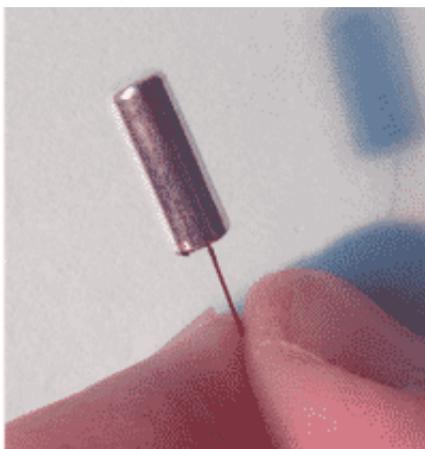
Tilt and impact sensor

The tilt and impact sensor is a simple switch which closes a circuit when positioned in a particular way. They are very cheap, around \$5 on eBay for a pack of 10. They come in a variety of shapes, but usually they are made of a tiny metallic cylinder with a thin copper wire coming out of one end.

The cylinder contains wires or a metal ball. When the device is hit or when its orientation changes, the wires or the ball come in contact with the wall of the cylinder and closes a circuit between the cylinder and the external wire.

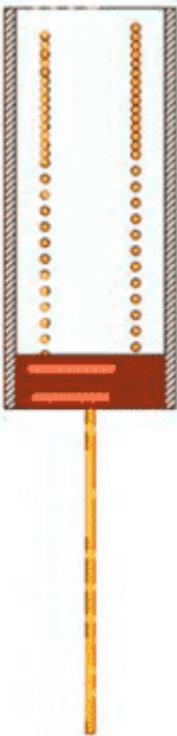
They can be very sensitive, so care must be given to compensating for this sensitivity, otherwise we would be getting readings that look chaotic. We are going to ignore this sensitivity for now.

Here is what a tilt sensor looks like from the outside...



Cylindrical tilt and impact sensor

... and on the inside.

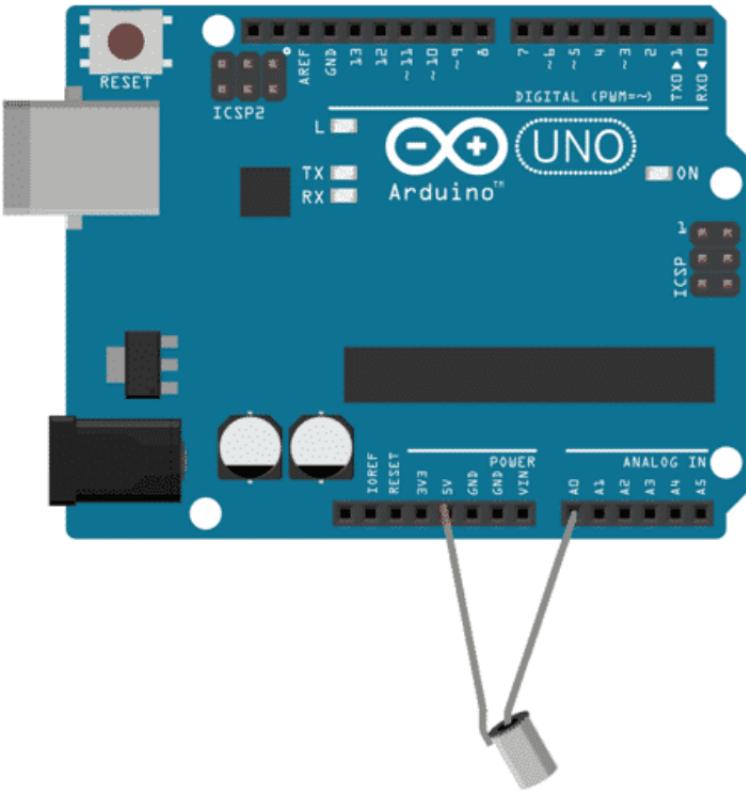


Assembly

To put this circuit together, we'll just use the Arduino, no breadboard is required.

We need to do a bit of soldering in order to connect the sensor to wires which we can connect to the Arduino. If you have never done soldering before, follow this [tutorial](#) for some instructions. Be careful not to get burned!

Here's what we are going to build.



Connect one wire to 5V and the other wire to A0 analog pin

Sketch

This is another very simple sketch. We'll just use the analog pin to determine if the switch inside the sensor is closed or open. You could just as easily have used a digital pin since there are only two states to detect, open or close, which nicely translate to HIGH or LOW.

[Here is the sketch on Github.](#)

```
int out;void setup(){Serial.begin(9600); // sets the serial port to 9600}void loop(){out = analogRead(0); // read analog input pin 0Serial.println(out, DEC);delay(100); // wait 100ms for next
```

reading}

Try out by connecting the sensor to the Arduino and moving the sensor in different directions. This is easier if you use some electrical tape to keep the sensor wires tidy. You could even stick the sensor onto the Arduino, and move the whole assembly instead of only the sensor. This will help in keeping the connections firm.

Ready for some serious Arduino learning?

Start right now with Arduino Step by Step Getting Started

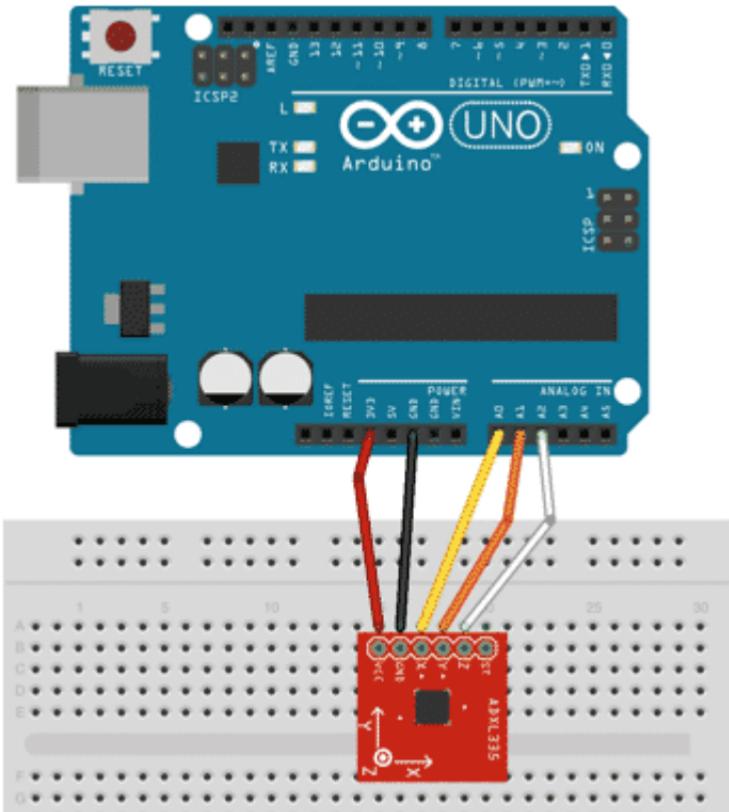
This is our most popular Arduino course, packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up.

Acceleration sensor

Arduino Sensors & Actuators guide series

Measuring acceleration

Acceleration is defined as the rate by which the velocity of an object changes. Having the ability to measure acceleration is very useful. For this purpose, we use an accelerometer.



Acceleration is defined as the rate by which the velocity of an object changes. The velocity of an object changes when a

force is applied to it. Acceleration is quantified by a direction and a magnitude. Direction is the way to which a force is directing the object, and magnitude relates to the strength of the applied force. Acceleration is described by [Newton's Second Law](#).

How can we measure acceleration?

Having the ability to measure acceleration is very useful. For this purpose, we use an accelerometer. An accelerometer measures the force that is applied to a small test mass. This test mass is placed inside the device and is held in place by one or more springs (or something equivalent). As gravity, or other forces, are applied on the test mass, they make it move towards a particular direction. The device measures the distance this mass travels from its resting position. The longer the distance, the stronger the force.

Imagine the accelerometer (or your self) in free fall, the test mass, as it is falling, is “feeling” no force being applied upon it. In a free-fall situation, the accelerometer would report no acceleration at all.

Accelerometers are everywhere

We use accelerometers in cars to detect collisions (and deploy airbags), or to gather performance statistics.

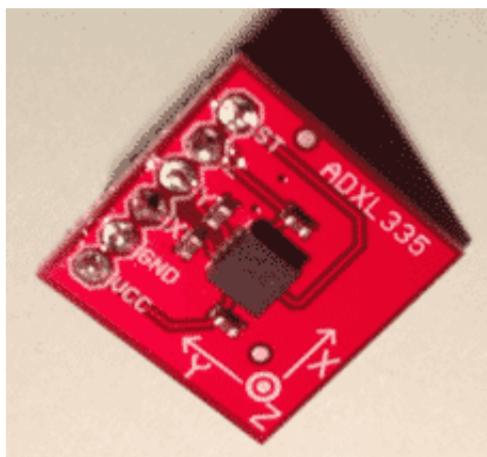
Accelerometers are embedded in smart phones to provide orientation information, and for making games that are played by moving a controller device.

You could use one in a toy car so that the car’s wheels stop spinning if it has turned over, or to make toys that react to the movement of a controller in 3-D space.

Accelerometers are also part of Inertial Navigation Systems (INS) that help vehicles (ships, planes, cars, submarines etc.)

maintain knowledge of their location when other positioning systems, like GPS, are not available.

In our experiment, we will use a common 3-axis accelerometer, the ADXL335. It is very cheap at around \$7 on eBay, and very easy to connect to the Arduino through any of the analog pins. It reports acceleration in 3 dimensions, X, Y and Z. It is very robust as it can detect forces of up to 10,000 Gs (1 G is equivalent to the Earth's gravity at the surface).



Assembly

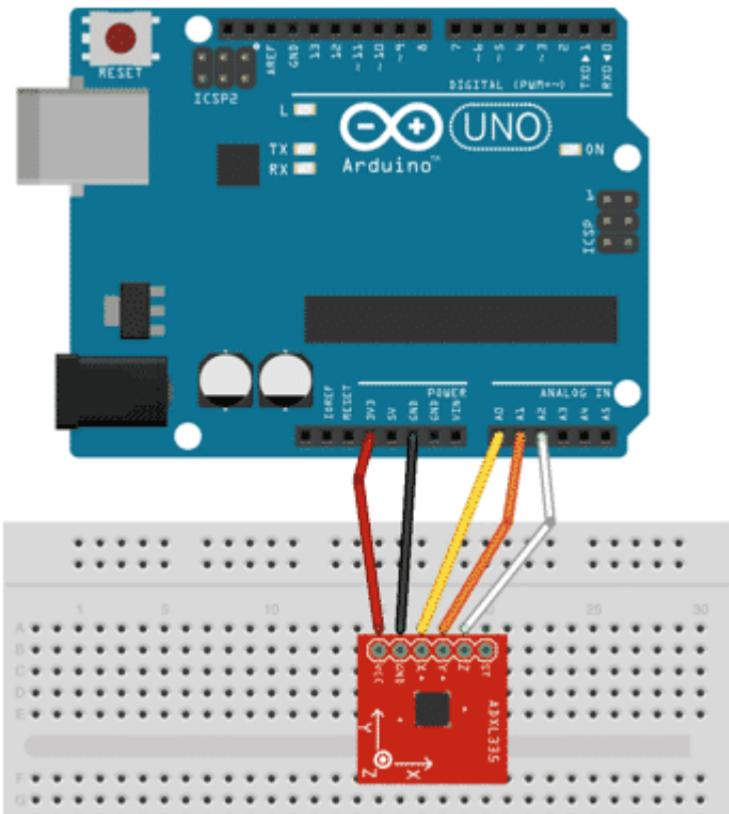
Let's put together this circuit and test out the motion sensor.

We will need:

- An Arduino
- Five jumper wires
- An ADXL335 3-axis accelerometer

The sensor takes three acceleration measurements, X, Y, and Z. The Arduino reads the analog outputs 0, 1, and 2, acquires the measurements, and prints them to the monitor.

Also, be very mindful of the power limits of this device. It needs 3.3V input, not 5V! If you provide 5V you may actually damage it!



Sketch

Done with the assembly, let's work on the sketch now ([here is the sketch on Github](#)).

```
int x, y, z; void setup() {Serial.begin(9600); // sets the serial port to 9600} void loop() {x = analogRead(0); // read analog input pin 0 y = analogRead(1); // read analog input pin 1 z = analogRead(2); // read analog input pin 1 Serial.print("accelerations are x, y, z: "); Serial.print(x, DEC); //
```

```
print acceleration in the X axisSerial.print(" "); // prints a space  
between the numbersSerial.print(y, DEC); // print acceleration  
in the Y axisSerial.print(" "); // prints a space between the  
numbersSerial.println(z, DEC); // print acceleration in the Z  
axisdelay(100); // wait 100ms for next reading }
```

Lets see how it works. `analogRead()` reads the voltage present at one of the analog pins. The accelerometer outputs voltage to its three outputs relevant to the forces applied to it.

For example, at rest, the reading on the Z axis is around 410. If I push the device upwards, I apply force along the Z axis which makes the test weight inside the accelerometer "feel" heavier, and the voltage on the Z pin increases, causing the reading to increase (i.e. to ~470, in my experiment). If I push the device downwards, the opposite happens.

By taking and evaluating measurements in all three axis, you could make it possible for your device to know the precise direction of its movement and the force applied to it.

Ultrasonic distance sensor

Arduino Sensors & Actuators guide series

Measure the distance to another object with the ultrasonic sensor

There are many types of technologies that can be used to make a proximity sensor. In this article, we focus on the ultrasonic sensor, which is essentially a “land sonar”: it emits a high frequency sound, far beyond what the human ear can hear, and waits for the echo.



There are lots of applications where we not only need to know that an object, or a person, is nearby, but also how far they are.

Imagine a robot moving around in a room. The robot can use a

distance (or proximity) sensor to detect that it is approaching a wall or another object. Or, you could use a proximity sensor to automatically open a door if a person is within a meter of the sensor.

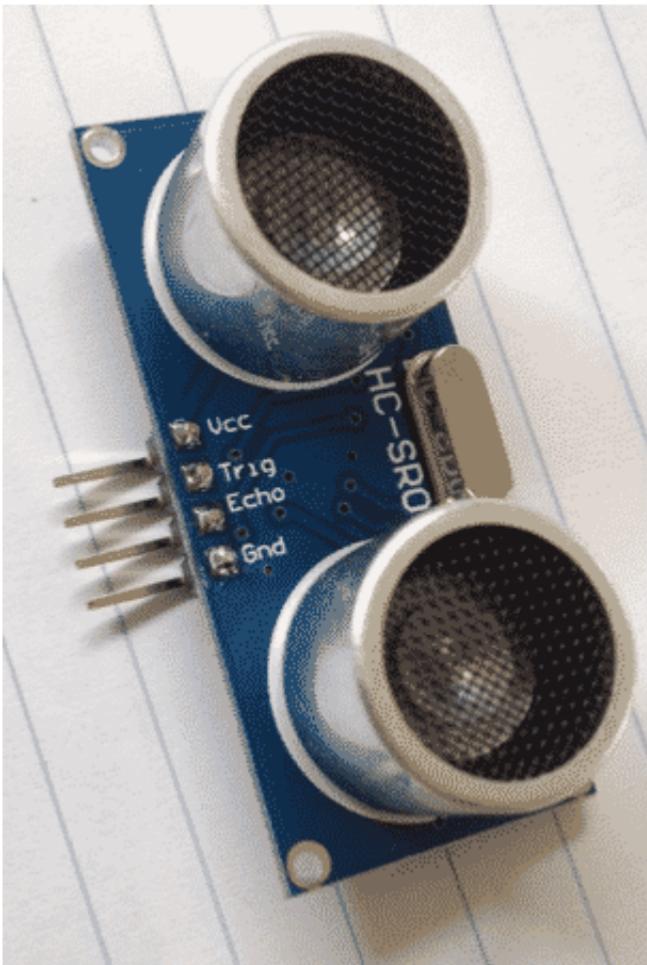
You find such sensors in cars (to help with parking and to avoid small accidents), and in smart phones where the smartphone can detect, for example, that the phone is held against the user's head, useful so that the screen is turned off to avoid accidental touchscreen input.

The ultrasonic sensor

There are many types of technologies that can be used to make a proximity sensor.

In this lecture we will focus on the ultrasonic sensor, which is essentially a "land sonar": it emits a high frequency sound, far beyond what the human ear can hear, and waits for the echo.

Once it captures the echo, it counts the time that elapsed between the emission of the ultrasound signal and the capture of its echo, and based on that it calculates the approximate distance of the object that produced the echo.



Ultrasonic sensors are solid-state devices, very reliable and cheap. Especially in indoor environments, and for small spaces (or measuring small distances), these sensors represent a good choice. Anything that is solid enough to allow sounds to bounce will work with these sensors.

If you want to measure or detect things like smoke and clouds, you will need to use something else, perhaps a microwave doppler radar.

For the Arduino, a commonly used proximity sensor is the HC-

SR04. You can find them on Ebay for less than \$2 each.

They are very easy to use, let's have a look.

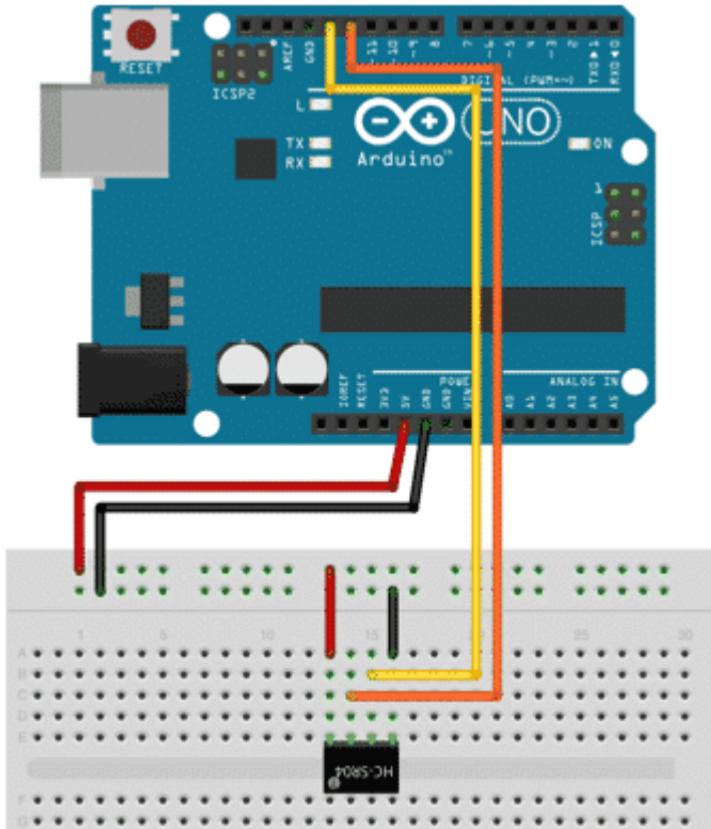
Assembly

Let's put together this circuit and test out the motion sensor.

We will need:

- The Arduino
- Six jumper wires
- An ultrasonic sensor, like the HC-SR04

Here's what we are going to build (as shown in the schematic below):



The sensor will constantly take distance measurements of whatever happens to be in front of it: your hand, books etc. The Arduino will receive these readings and print them to the Serial monitor. Very simple.

Sketch

Done with the assembly, let work on the sketch now ([here is the sketch on Github](#)).

```
#define trigPin 13#define echoPin 12void setup() {Serial.begin(9600);pinMode(trigPin, OUTPUT);pinMode(echoPin, INPUT);}void loop() {long duration, distance;digitalWrite(trigPin,
```

```
LOW);delayMicroseconds(2);digitalWrite(trigPin,  
HIGH);delayMicroseconds(10);digitalWrite(trigPin,  
LOW);duration = pulseIn(echoPin, HIGH);distance =  
(duration/2) / 29.1;if (distance >= 200 || distance <=  
0){Serial.println("Out of range");}else  
{Serial.print(distance);Serial.println(" cm");}delay(500);}
```

There's quite a lot happening in this small amount of code.

We define the sensor's trigger and echo pins to be 13 and 12 respectively. In the `setup()` function, we initialize the Serial monitor, and set pin 13 to be the output and pin 12 to be the input.

Through pin 13, the Arduino will ask the sensor to trigger a ping, similar to the "boing" noise that submarines emit when they use their sonar. This ping, assuming it bounces off an object in range, will come back and will be picked up by the sensor's receiver. The Arduino will know when that happens because it is monitoring pin 12, which is connected to the sensor's echo pin.

In the `loop()` function, we first setup two variables of type `long`. Long numbers are 4 bytes in size, a total of 32 bits, and can hold very large numbers: -2,147,483,648 to 2,147,483,647. The variable `duration` will hold the total number of microseconds that it took for the ping to reach the object and return to the sensor. The variable `distance` will contain the distance to that object in centimeters.



The Arduino is triggering a ping by writing to the trigger pin three pulses: first, a digital LOW for 2 microseconds, then a digital HIGH for 10 microseconds and finally a digital LOW which stays low until the next iteration of the loop.

It then uses the function `pulseIn()` to get the number of microseconds in takes of the ping to come back. `PulseIn()` accepts two parameters: a pin number (in our case it is 12, stored in variable `echoPin`), and the pulse level we want to detect, in our case it is HIGH because we want to detect the 10 microsecond ping we just emitted. As soon as the Arduino calls the `pulseIn()` function, it starts timing. It returns the number of microseconds from the time the function was called until it detects the ping echo.

How Arduino calculates the distance

The distance is calculated by the Arduino. It divides by two the duration that the `pulseIn()` function returned, since the ping travels a total of twice the distance to the object (going there and its echo coming back). It then divides again by the “magic number” 29.1. This number derives from this calculation:

The speed of sound at 0 degrees celsius is measured to be 331.5 meters per second. At different temperatures, the speed of sound is calculated by adjusting 331.5m/s for the temperature by multiplying by 0.6:

$$\text{SpeedOfSound}(\text{Temperature}) = 331.5 + 0.6 * \text{Temperature}$$

At 20 degrees, this works out to be 343.5 m/s.

We need to convert the seconds to microseconds and the length from meters to centimetres:

$$\text{SpeedOfSound} = 343.5 * 100 / 1,000,000 = 0.03435 \\ \text{cm/microseconds}$$

This means that sound, at 20 degrees, can travel a distance of 0.03435 centimetres in one microsecond. If a signal and its echo take X microseconds to do the round trip, then the total distance covered is:

$$\text{Total_distance} = X * 0.03435 = X / 29.1$$

We adjust this so that we only include the duration of the one-way trip to the target (instead of the return trip), and the formula becomes:

$$\text{distance} = (X / 2) * 0.03435 = (X / 2) / 29.1$$

Wikipedia has a [very good article](#) on how to calculate the speed of sound, for the curious.

Limitations

If the distance to a target is over 200 centimeters, the Arduino reports that the target is out of range, since at that distance measurements are not reliable.

The same happens if the distance is negative

Question to consider: why do we need to test for negative distance?

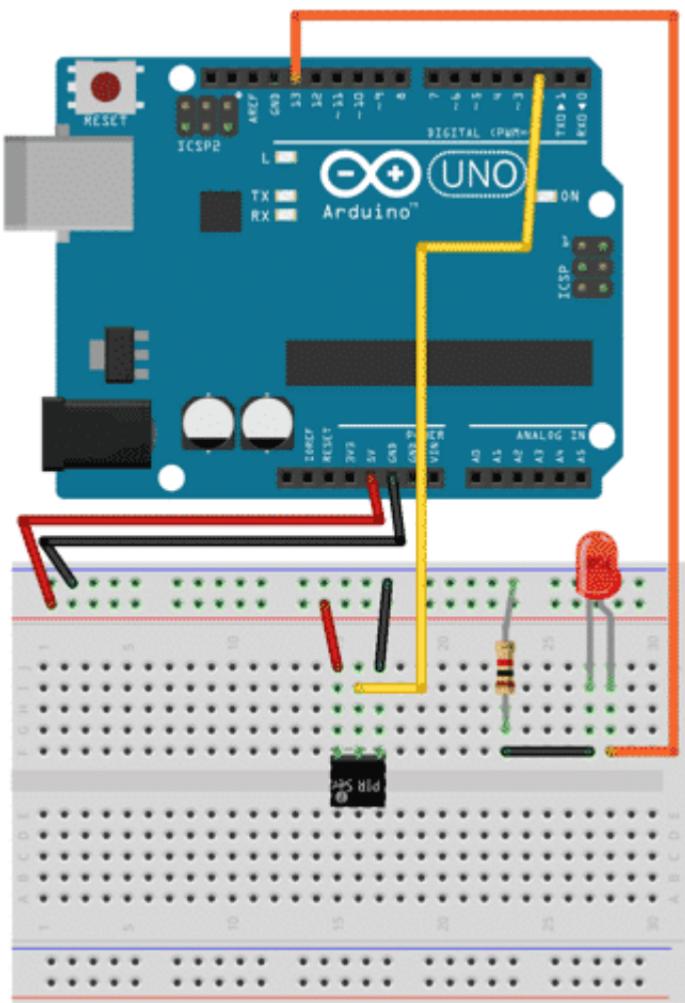
Any other distance condition is valid, so the monitor will print out the distance in centimeters.

PIR sensor

Arduino Sensors & Actuators guide series

Detect motion with the PIR sensor

Passive infrared (PIR) sensors are very common and typically found in home electronics. They detect the heat that is emitted by the body of a person in a room as it contrasts against the background heat.



Knowing if something is moving is useful in many applications.

The classic example is security, where an alarm system can detect an intruder moving inside a room, so that it can notify the police.

Another common use is in home and office automation, where you could get the lights to turn on and off automatically depending on whether someone is still in the room or turning

on the flood light in your driveway as your car approaches.

There are several technologies used in motion sensors, each with their own capabilities and price points.

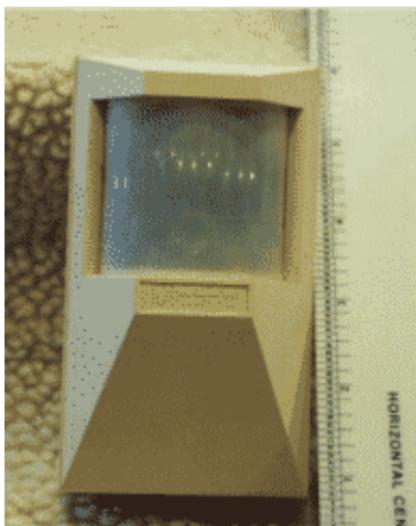
PIR sensors

Passive infrared (PIR) sensors are very common and typically found in home electronics.

They detect the heat that is emitted by the body of a person in a room as it contrasts against the background heat.

A PIR sensor does not emit any energy, it just sits there and waits for a heat source to enter its field of vision.

They look like this (an example of a home security system motion sensor - below).



Ultrasonic distance sensors

An ultrasonic motion sensor uses ultrasounds to detect moving objects.

Just like bats, an ultrasonic sensor emits ultrasounds at frequencies from 30khz to 50kHz and then picks up their echo.

These sensors can often measure the time a signal takes to return, and from that it can calculate the distance to the object.

Therefore, ultrasonic sensors can calculate both distance from an object as well as whether the object is moving. We had a look at the ultrasonic distance sensor in the previous lesson.

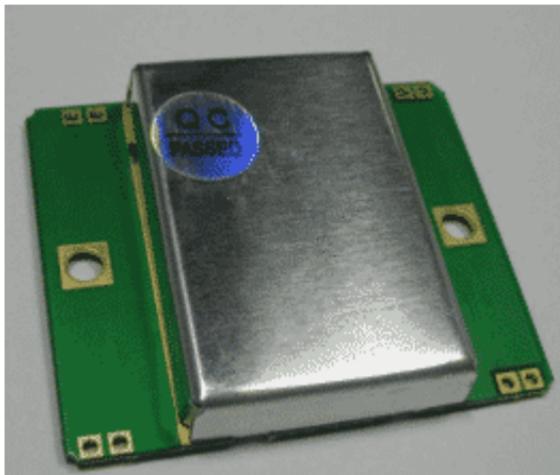


Microwave motion sensors

A microwave motion sensor works on the same principle as the ultrasonic sensor, except that instead of ultrasounds it emits microwaves. They are still relatively cheap, and because microwaves are much higher in terms of frequency than ultrasounds, motion can be detected with a lot more detail. Many microwave motion sensors can determine not only the

motion itself, but also distance and speed using the Doppler effect.

Here is what a microwave sensor module looks like (of course the module is covered by a plastic cover when installed – below).



In this lecture, we will connect a passive infrared sensor to our Arduino, calibrate it, and turn an LED on every time motion is detected.

Assembly

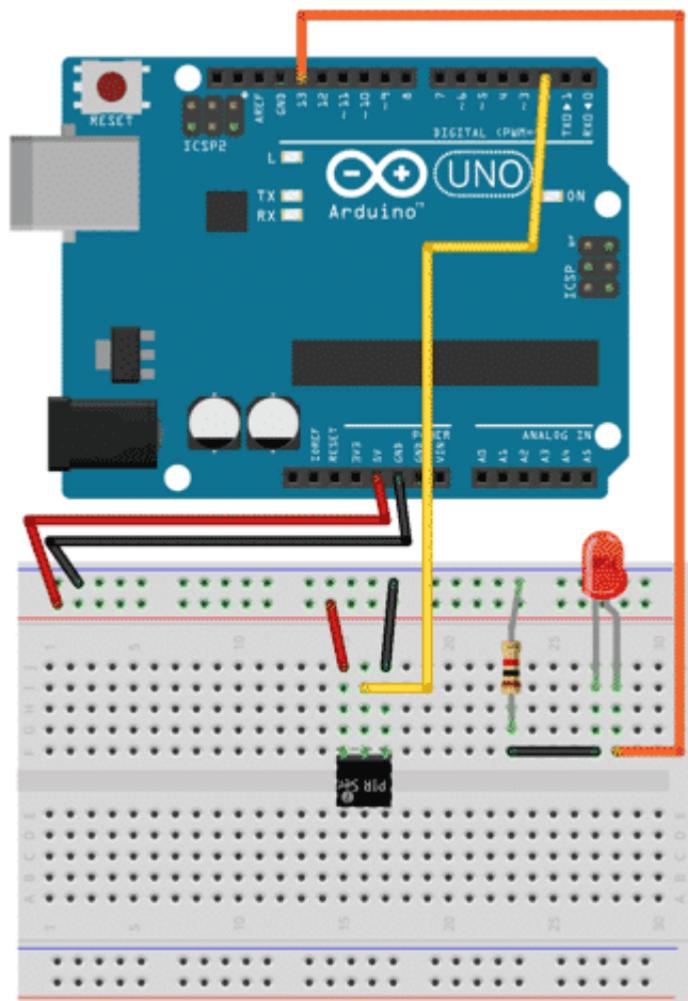
Let's put together this circuit and test out the motion sensor.

We will need:

- An Arduino
- Four jumper wires
- A PIR sensor, like the HC-SR501
- One resistor, 1k
- One LED

Here's what we are going to build.

While you are connecting the motion sensor, it's a good idea to remove the sensor cover so you can see the pin markings, ensuring that power is connected the right way.



This circuit will detect motion through the sensor, and send a signal to the Arduino via digital pin 2. The Arduino will receive the signal and in turn activate the LED via digital PIN 13.

Notice that the Arduino Uno board already has an small LED connected to digital port 13, so you could choose to not connect yours.

Sketch

Done with the assembly, let work on the sketch now.

[Here is the sketch on Github.](#)

```
/** PIR sensor tester*/int ledPin = 13; // choose the pin for the LED
int inputPin = 2; // choose the input pin (for PIR sensor)
int pirState = LOW; // we start, assuming no motion detected
int val = 0; // variable for reading the pin status
void setup()
{
  pinMode(ledPin, OUTPUT); // declare LED as
  outputMode(inputPin, INPUT); // declare sensor as
  inputSerial.begin(9600);
}
void loop()
{
  val = digitalRead(inputPin); // read input value if (val == HIGH) {
  // check if the input is HIGH
  digitalWrite(ledPin, HIGH); // turn LED ON if (pirState == LOW) {
  // we have just turned on
  Serial.println("Motion detected!"); // We only want to print // on
  // the output change, not state
  pirState = HIGH;
} else
{
  digitalWrite(ledPin, LOW); // turn LED OFF if (pirState == HIGH) {
  // we have just turned off
  Serial.println("Motion ended!"); // We only want to print on // the output change, not state
  pirState = LOW;
}
```

By now, this sketch should be easy to read and understand.

We start by setting constants for the pins and values.

The LED is connected to digital pin 13, and the sensor's output to digital pin 2.

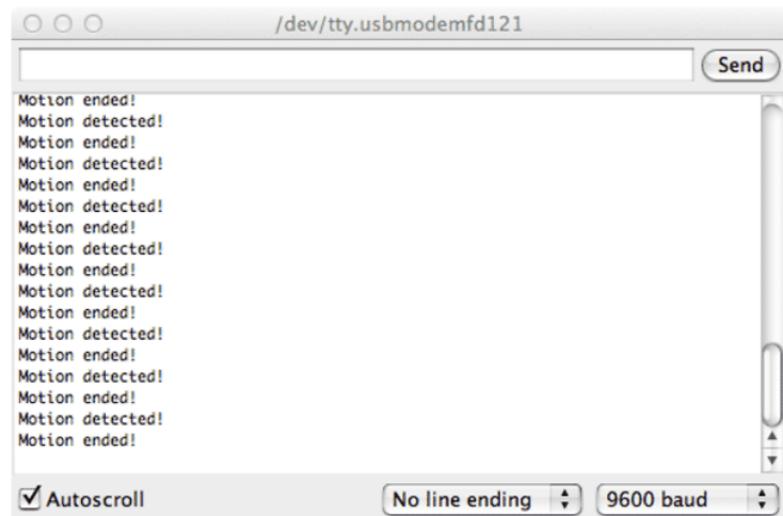
We also assume that when the Arduino starts, there is no motion, so variable pirState is set to LOW, and val, the variable to which the output state of the PIR sensor is stored, is 0 (LOW).

In the setup() function, we set pin 13 to be output, and pin 2 to be input. We also initialize the serial port so that we can see text output in the monitor window.

In the loop() function, we constantly read the value of the PIR sensor by using the digitalRead(inputPin) function. This function reads voltage in the range of 0V to 3.3V (at least for the sensor I am using), and the Arduino translates that to LOW and HIGH respectively.

If HIGH is detected, the Arduino will set pin 13 to HIGH and this will activate the LED. If the previous state of the sensor was LOW, then the Arduino detects this as new motion, so it will print a message to the monitor, and set pirState to HIGH. This will prevent the Arduino from continuously printing out that new motion was detected while the actual motion is still continuing.

Upload it to see it working, don't forget to open up the monitor window (Tools > Serial Monitor). You should see something like this:



The screenshot shows the Arduino Serial Monitor window. The title bar says "/dev/tty.usbmodemfd121". The main text area contains the following text:
Motion ended!
Motion detected!
Motion ended!
Motion detected!

At the bottom of the window, there are three buttons: "Send" (with a small icon), "Autoscroll" (which is checked), and two dropdown menus for "No line ending" and "9600 baud".

If you are not sure what the pirState variable is actually doing, do this little experiment:

Modify the sketch by replacing the lines:

```
if (pirState == LOW) { Serial.println("Motion detected!");  
pirState = HIGH;}
```

with only:

```
Serial.println("Motion detected!");
```

... and the lines:

```
if (pirState == HIGH){ Serial.println("Motion ended!"); pirState  
= LOW;}
```

... with only:

```
Serial.println("Motion ended!");
```

Upload the edited sketch.

Open the monitor and activate the sensor by waving your hand above it. What can you see in the monitor?

You can calibrate the sensitivity and amount of time that the sensor stays activated by turning the two small orange knobs.

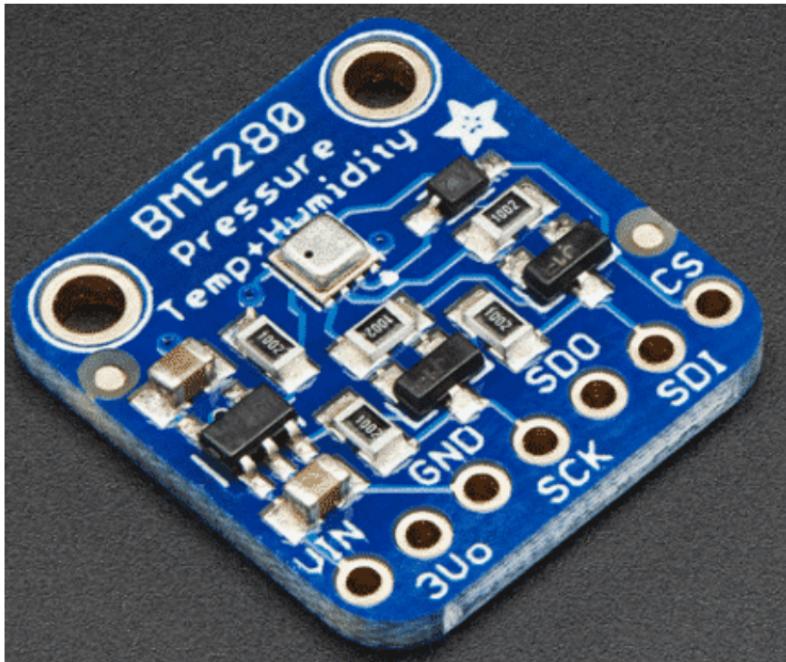
Experiment with them to see the effect they have on the sensor's output.

BME280

Arduino Sensors & Actuators guide series

Temperature and barometric sensor BME280

Atmospheric pressure is defined as the weight of a column of air above an object. Measuring the atmospheric pressure has several applications. We will be measuring atmospheric pressure by using the BME280 sensor.



Measuring the atmospheric pressure has several applications.

Obviously, if you are a meteorologist, knowing the pressure at a geographical location helps forecasting the weather. But there's more.

Atmospheric pressure is defined as the weight of a column of air above an object.

As the height of a column of air above an object changes depending on its altitude, so does the weight of that column.

Atmospheric pressure at the surface of the sea is higher than that on the top of a tall mountain because the column of air above it is higher. Therefore, measuring the atmospheric pressure is also a simple way of figuring out your altitude, or the altitude of one of your flying gadgets.

The standard unit for measuring atmospheric pressure is "Pa", or "Pascals". At sea level, the standard pressure is defined to be 101.325 kPa, or 101,325 Pa.

The BME280 sensor

We will be measuring atmospheric pressure by using the BME280 sensor. This sensor costs around \$8 on eBay. It can measure pressure from 300hPa to 1100hPa, which converts to around 500 meters below sea level to 9,000 meters above sea level.

It's accuracy is also excellent, around 0.03hPa. "hPa" is pronounced "hectoPascal".

Another nice thing about this sensor is that it also measures the temperature.

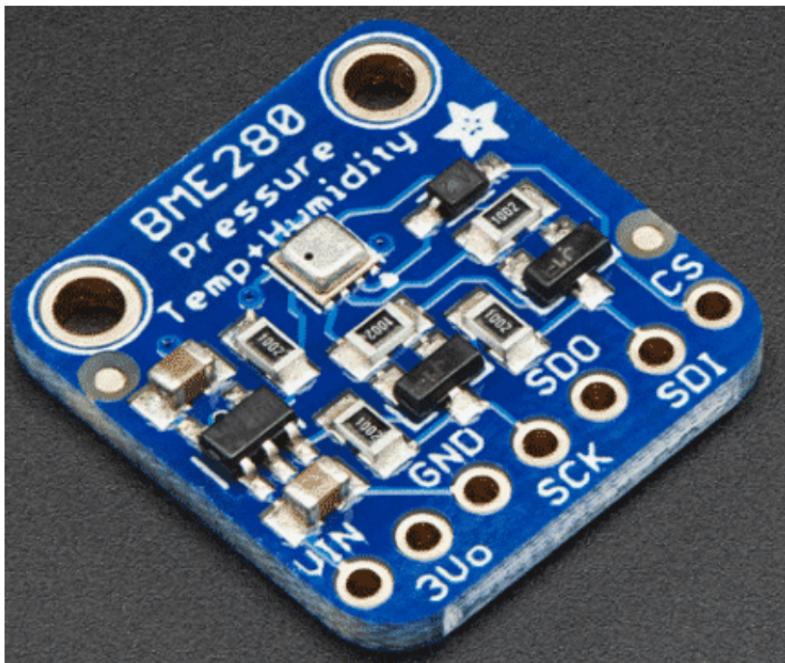
The BME280 talks to other devices via the I2C interface, a digital serial communications interface that only needs two wires for communication, and two for power.

One communication wire is called SDA, and it transmits data,

while the second, SCL, is for the clock signal.

A clock signal is needed because I2C is a synchronous interface.

The sensor uses 3.3V or 5V, which the Arduino conveniently provides. Here's what this sensor looks like (below).



The BME280 measures pressure by taking advantage of the piezo-resistive property that silicon and germanium have.

This property involves the change in resistance in those materials depending on the amount of mechanical load that is put on them.

Assembly

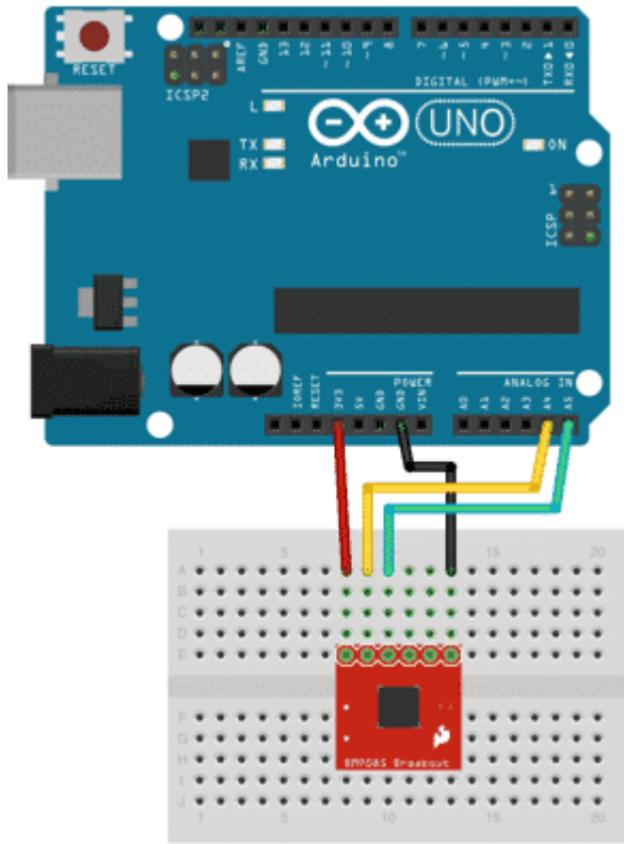
Let's put together this circuit and try out the sensor.

We will need:

- An Arduino
- Four jumper wires
- A BME280 sensor device

This is very simple wiring using the I2C protocol:

- Connect Vin to the power supply, 3-5V is fine. Use the same voltage that the microcontroller logic is based off of. For most Arduinos, that is 5V
- Connect GND to common power/data ground
- Connect the SCK pin to the I2C clock SCL pin on your Arduino. On an UNO, this is also known as A5,
- Connect the SDI pin to the I2C data SDA pin on your Arduino. On an UNO, this is also known as A4.



Sketch

Done with the assembly, lets work on the sketch now.

To begin reading sensor data, we need to [install the Adafruit_BME280 library](#). It is available from the Arduino library manager so we recommend using that.

From the IDE open up the library manager following the menu Sketch/Include Library/ Manage Libraries And type in adafruit bme280 to locate the library. Click Install and wait until it is finished.

Also add the Adafruit Unified Sensor library the same way.

Here's the sketch, it comes straight of the examples that are included with the Arduino IDE. I have added some comments to help you understand what is going on ([here is the sketch on Github](#)):

```
#include /*Include the Wire library which allows to use the I2C
interface*/#include #include /*library to easily take readings
from the sensor*/#define SEALEVELPRESSURE_HPA
(1013.25)Adafruit_BME280 bme; /*Declare the bpm variable,
an easy to remember reference for the device*unsigned long
delayTime;void setup() { Serial.begin(9600); /*Setup serial
communication and speed* Serial.println(F("BME280 test"));
bool status; status = bme.begin(); /*Try to start the device* if
(!status) { /*If it is not starting, print message*
Serial.println("Could not find a valid BME280 sensor, check
wiring!"); while (1); /* Go in an endless loop. This prevents the
Arduino from calling the loop function* } Serial.println("-"
Default Test -"); delayTime = 1000; Serial.println();}void
loop() { printValues(); delay(delayTime);}void printValues() {
Serial.print("Temperature = "); /*Read and print temperature*
Serial.print(bme.readTemperature()); Serial.println(" *C");
Serial.print("Pressure = "); /*Read and print pressure*
Serial.print(bme.readPressure() / 100.0F); Serial.println(" "
hPa"); // Calculate altitude assuming 'standard' barometric
pressure of 1013.25 millibar = 101325 Pascal
Serial.print("Approx. Altitude = ");
Serial.print(bme.readAltitude(SEALEVELPRESSURE_HPA));
/*Read and print altitude*/ you can get a more precise
measurement of altitude if you know the current sea level
pressure which will vary with weather and such. If it is 1015
millibars that is equal to 101500 Pascals. Serial.println(" m");
Serial.print("Humidity = "); Serial.print(bme.readHumidity());
/*Read and print humidity* Serial.println(" %");
Serial.println();}
```

Now we'll run this sketch and look at the monitor output:

And that is how you connect and use the BME280 barometric

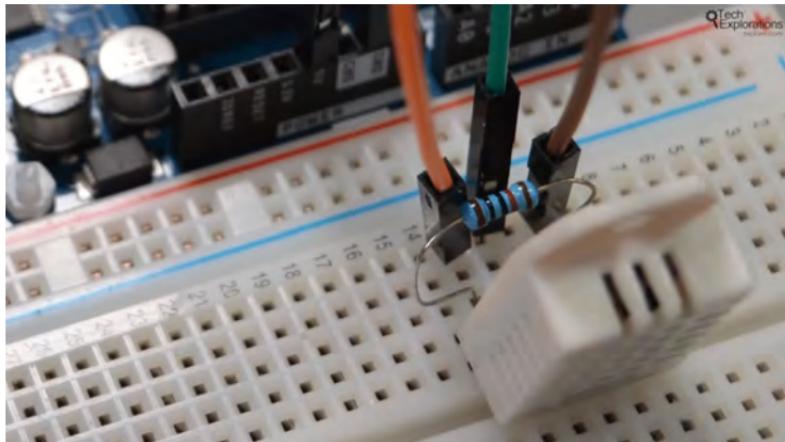
sensor with the Arduino!

Measuring temperature and humidity

Arduino Sensors & Actuators guide series

Measuring temperature and humidity

The DHT22 is a versatile, affordable, and reliable sensor that is widely used for measuring temperature and humidity in various projects. This guide provides information on the sensor's operational parameters, wiring instructions, and a sketch for extracting temperature and humidity readings.



Introduction

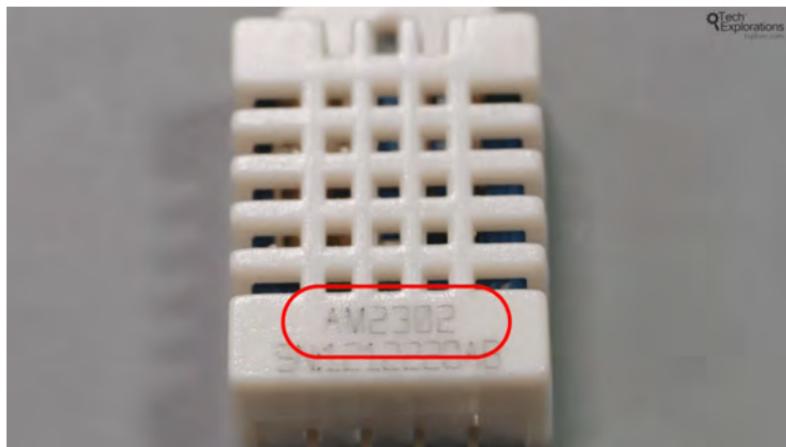
In this article, we will explore the DHT22 (or DHT11) sensor, which is widely used for measuring temperature and humidity. This versatile digital sensor provides accurate readings without the need for additional calculations or conversions. Let's dive into the details and learn how to use this sensor effectively.



The DHT22 sensor

Understanding the DHT22 Sensor

The DHT22 sensor, also known as the AM2302, operates within a power supply range of 3.3 to 6 volts DC. This flexibility allows it to be used with various Arduino boards. It can be used with 3.3V Arduino boards like the Arduino Due or Arduino Pro Mini. Just connect it to the 3.3V pin. If you're using it with the 5V Arduino Uno, connect it to the 5V pin.



The DHT22 sensor, also known as the AM2302

The sensor's operational parameters indicate that it can measure temperatures ranging from -40 to 80 degrees Celsius and relative humidity from 0 to 100 percent. With an accuracy

of ± 0.5 degrees Celsius for temperature and ± 2 percent for relative humidity, the DHT22 provides reliable results for most applications. In case you require higher precision, I have another article that introduces the BMP180, a more accurate environmental sensor for temperature and humidity. Additionally, the DHT22 offers a resolution of 0.1 percent for relative humidity and 0.1 degrees Celsius for temperature.

3. Technical Specification:

Model	DHT22
Power supply	3.3-6V DC
Output signal	digital signal via single-bus
Sensing element	Polymer capacitor
Operating range	humidity 0-100%RH; temperature -40~80Celsius
Accuracy	humidity $\pm 2\%$ RH(Max $\pm .5\%$ RH); temperature ± 0.5 Celsius
Resolution or sensitivity	humidity 0.1%RH; temperature 0.1Celsius
Repeatability	humidity $\pm 1\%$ RH; temperature ± 0.2 Celsius
Humidity hysteresis	$\pm 0.3\%$ RH
Long-term Stability	$\pm 0.5\%$ RH/year
Sensing period	Average: 2s
Interchangeability	fully interchangeable
Dimensions	small size 14*18*5.5mm; big size 22*28*5mm

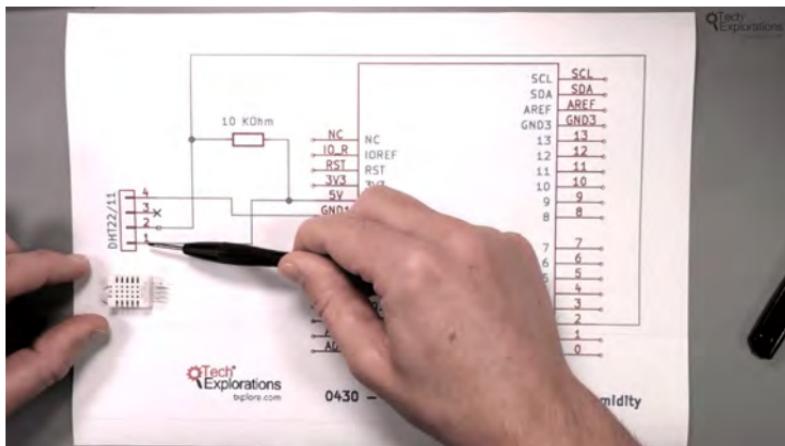
The sensor's datasheet provides information about its operational parameters.

It is important to note that the DHT22 sensor operates at a slow speed, taking about two seconds to provide a reliable reading to the microcontroller. Therefore, avoid making frequent calls to the sensor for new values.

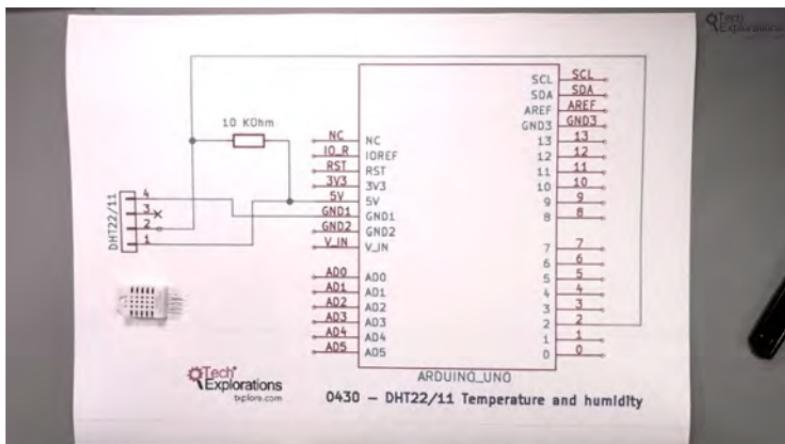
Wiring

Wiring Diagram

Below you can see the wiring diagram. Make sure to orient the sensor with the grill facing you, not the back, to ensure the correct pin numbering.



Pin number one is located on the leftmost end of the sensor, while pin number four is on the rightmost end. In some cases, you may come across DHT22 sensors with only three pins. These are typically sold as part of a breakout for the sensor. In these cases, the number three pin is left unconnected. You can simply push it up and set it aside.



Wiring the DHT22 Sensor

To wire the DHT22 sensor to an Arduino board, follow these simple steps:

1. Connect pin number 1 of the sensor to the 5-volt pin on the Arduino (or the 3.3-volt pin for a 3.3-volt Arduino).
2. Connect pin number 4 of the sensor to the ground pin on the Arduino.
3. Connect pin number 2 of the sensor (the data pin) to a digital pin on the Arduino (e.g., digital pin number 2). You can configure it to use any of the available digital pins on your Arduino.
4. Include a 10 kilohm pull-up resistor between pin number 2 and the 5-volt pin. This resistor ensures a stable voltage when the sensor is not transmitting data.

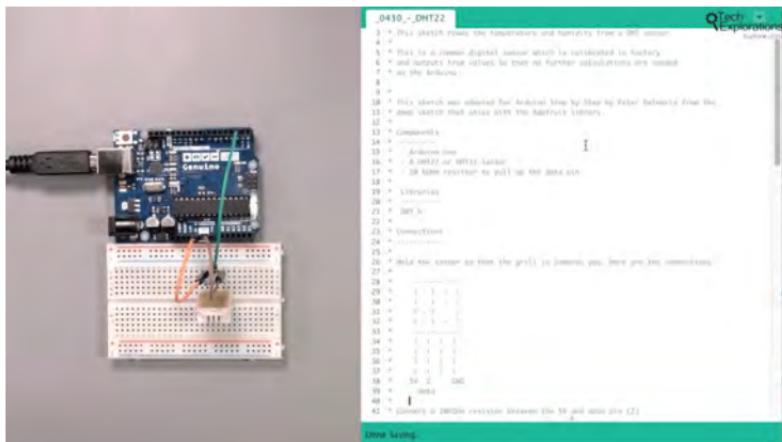
Pull-Up Resistors

A pull-up resistor is used to pull up the voltage at pin number two to five volts. This ensures a defined voltage when the DHT sensor is not transmitting any data, preventing a condition called floating. In this case, a strong pull-up resistor with a value of 10 kilohms is used, but pull-up resistors can range from 10 kilohms to 50 or 100 kilohms, or even higher.

To better understand the concept of pull-up and pull-down resistors, you can refer to my article [here](#). It provides background information and a deeper understanding of the role of this resistor.

Sketch

Below is the sketch that provides some information about the connections as well.



```
[tcb-script
src="https://emgithub.com/embed-v2.js?target=https%3A%2F%2Fgithub.com%2Ffutureshocked%2FArduinoSbSGettingStarted%2Fblob%2Fmaster%2F_0430_-_DHT22%2F_0430_-_DHT22.ino&style=night-owl&type=code&showFullPath=on&fetchFromJsDelivr=on"][/tcb-script]
```

Extracting Temperature and Humidity Readings

To extract temperature and humidity readings from the DHT22 sensor, we use the [Adafruit DHT library](#). This library supports various DHT sensors, including the DHT22, and provides functions for reading the temperature and humidity values from the sensor.

```
#include "DHT.h"
```

Within the Arduino sketch, specify the sensor type (DHT22) and the data pin connected to the sensor (e.g., digital pin number two, or you can adjust it to whichever digital pin you have available).

```
#define DHTTYPE DHT22
```

```
#define DHTPIN 2
```

In line 76 of the sketch, the C language dht object is created. The parameters for the data pin and type are passed in.

```
#define DHTPIN 2
```

In the setup() method, the serial monitor is started and a message is printed to begin. Then, the sensor is started. In the loop(), there is a two-second delay between subsequent measurements, as stated in the datasheet.

Remember to include a two-second delay between subsequent measurements to allow the sensor to provide accurate readings.

After the two-second delay, the humidity measurement is obtained by calling the readHumidity() function on the dht object, and the result is stored in a floating-point variable called h. The same process is repeated for temperature with the dht.readTemperature() function.

By default, the library provides temperature readings in Celsius. To get readings in Fahrenheit, the true parameter must be passed into the readTemperature() function. This eliminates the need to create a custom function for Celsius to Fahrenheit conversion.

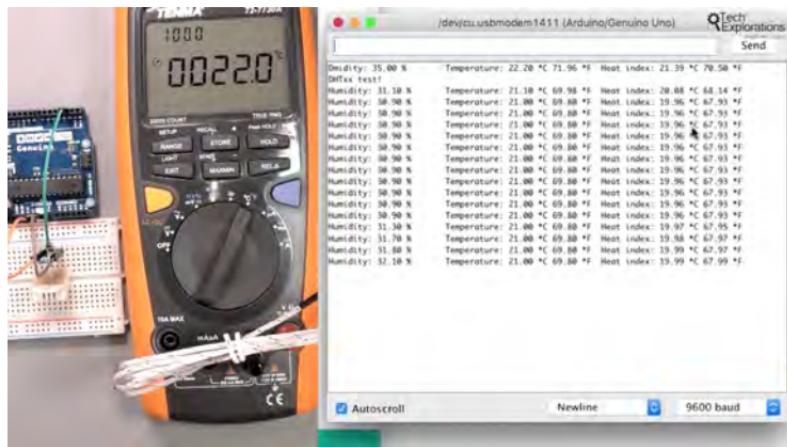
Next, a check is performed to ensure that the sensor is functioning properly. If the sensor is malfunctioning, the values obtained may not be actual numbers and could be something like “not available” or garbage. The function isnan() is used to check if the values are valid numbers before printing them on the serial monitor.

The results for humidity and temperature in Fahrenheit or Celsius are printed with Serial.print(). The computeHeatIndex() function calculates the heat index, which is the perceived

temperature based on the combination of actual temperature and humidity. The library internally calculates the heat index using the humidity and temperature readings. If you want to learn more about the heat index, you can refer to the relevant [Wikipedia article](#). The [table](#) in the article shows examples of how the perceived temperature differs from the measured temperature based on relative humidity.

Uploading the Sketch

All right, let's upload the sketch and see the hardware in action. Currently, the humidity in my lab is around 30 percent, the temperature is 21 degrees Celsius, and the heat index is 19 percent. Due to the low humidity, the apparent temperature is slightly lower than the actual temperature.



Testing the accuracy of the DHT22 with a multimeter

To test the accuracy of the DHT22, I can use my multimeter. According to my multimeter, the current temperature is 22 degrees Celsius, while the DHT22 is giving me a reading of 21 degrees Celsius. However, it's important to note that my multimeter is not perfect either. I don't have a calibrated temperature meter or thermometer specifically for lab use, so I can't determine which reading is more accurate. Nevertheless,

for most users, a slight difference of 21 to 22 degrees Celsius is not significant.

Interpreting the Results

Once you have obtained the temperature and humidity readings, you can use them for various purposes. If desired, you can calculate the heat index, which is particularly useful for understanding how humans perceive the temperature in a given environment. Keep in mind that the DHT22 sensor might not be as fast as other sensors, as it takes about two seconds to provide a reliable reading. Therefore, avoid querying the sensor too frequently to allow for accurate data collection.

Conclusion

The DHT22 sensor is an excellent choice for measuring temperature and humidity in various projects. Its simplicity, affordability, accuracy, and reliability make it a popular option among hobbyists and professionals alike.

Whether you are building an environmental monitoring system or simply curious about the weather conditions, the DHT22 sensor is a valuable tool to have in your electronics toolkit/highly recommended sensor for anyone who is developing an environment gadget.

If you have any questions or need further assistance, feel free to reach out. Happy sensing!

Ready for some serious Arduino learning?

Start right now with Arduino Step by Step Getting Started

This is our most popular Arduino course, packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up.