FROM THE AUTHOR WHO BROUGHT YOU
"MAKER EDUCATION REVOLUTION"

Tech Explorations

ARDUINO:
A STARTING UP GUIDE FOR COMPLETE BEGINNERS

Free Edition

BY DR PETER DALMARIS

A book designed to help you take your first steps in the amazing world of the Arduino and Making.

-- BLANK PAGE --

Arduino

A STARTING UP GUIDE FOR COMPLETE BEGINNERS

by Peter Dalmaris, PhD

This book is designed as a guide for people new to the Arduino platform.

It will help you understand the Arduino as a technology and platform, set it up on your computer, do your first experiments with hardware, and understand the role of the Arduino in the evolution of the Internet of Things.

This guide was written by Peter Dalmaris.

TABLE OF CONTENTS

COPYRIGHT

ABOUT THE AUTHOR

Peter Dalmaris is Chief Geek at Futureshock Enterprises Pty Ltd, expressing his interests in technology through Tech Explorations. Tech Explorations creates video courses and books on technologies such as the Arduino, and the Raspberry Pi. Peter also publishes a blog, and Stemiverse Podcast, in which explores and discusses topics in technology and education.

Peter's mission is to assist people of all ages in their technology exploration adventures.

His background is in Electrical and Electronics Engineering, and he has spent a lot of time as a software systems engineer. He has been working actively with the Arduino since 2007.

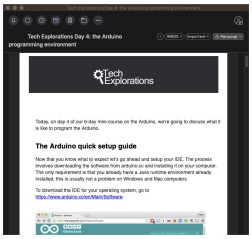
Peter has also been an educator since 2000, teaching at various Universities in Sydney, Australia.

Peter's technology interests include (but are not limited!) to microcontrollers, embedded systems, CAD, 3D printing, open-source software and programming languages, Internet of Things, web technologies, home automation, robots, remote sensing and much more.

You can contact Peter via Twitter (@futureshocked), his web site (techexplorations.com), and Facebook (facebook.com/txplore).

BOOK (FREE) COMPANION

This book is complemented by an email course.

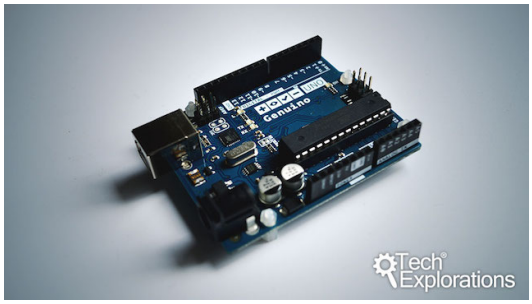


If you are a super-busy person and prefer to learn by studying for a few minutes every day, you will probably find our free 6-day email introduction to the Arduino course perfect. Each day (starting today), you will receive an email that contains a bite-sized lesson from the book and the video course. Have a quick read, get the gist, and move on with your day. When you are ready, dive in to this book or logon to your account at techexplorations.com to access your video course.

SOME OF OUR ARDUINO COURSES YOU MIGHT BE INTERESTED IN

We are seriously crazy about the Arduino, so we have created a lot of educational content. Here's a small selection to get your appetite going:

ARDUINO STEP BY STEP: GETTING STARTED



This course is for the new Arduino Maker. It is the perfect "next step" for learners who have completed this free introductory book and course.

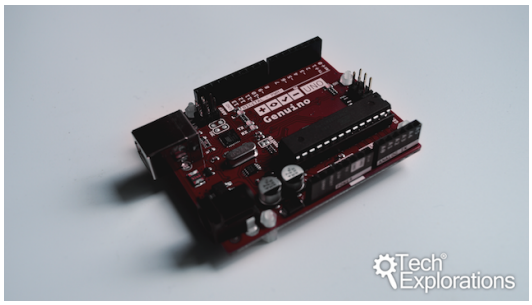
In making this course, I emphasized the importance of getting the basics right and learning to mastery. As an educator for over 15 years, I know first-hand that hitting a roadblock because you lack the fundamental knowledge to progress can be demotivating.

In ASbS: Getting Started, we make sure that in the more than 15 hours of video content, mini projects and quizzes, we cover all the basics so that you can enjoy learning about the Arduino.

By the end of the course, you will have a good understanding of the capabilities of the Arduino Uno, the best Arduino for people getting started, and you will be familiar with the capabilities of several of its cousins.

You can find more details at https://techexplorations.com/courses/asbs-getting-started/

ARDUINO STEP BY STEP GETTING SERIOUS



This course will take your Arduino skill to the next level. It will help you extend your knowledge of Arduino components and techniques and build up new skills in the largest, and the most comprehensive course on the Web!

Studying Arduino Step by Step Getting Serious is a very serious undertaking. The course is split into 40 sections and over 250 lectures spanning more than 30 hours of video content.

In each section, you will learn a specific topic.

Each topic contains:

- multiple examples of code
- wiring schematics
- demonstrations of a completed circuit
- alternative scenarios

Here are some of the topics that you will learn about in this course (for a full list, please look at the course curriculum):

- Advanced environment, motion, position and GPS sensors.
- New ways to receive input from the user with keypads, potentiometers, and encoders.
- New ways to provide feedback to the user, with color screens, complete with a touch-sensitive surface!
- Awesome ways to configure LEDs, monochrome or color.

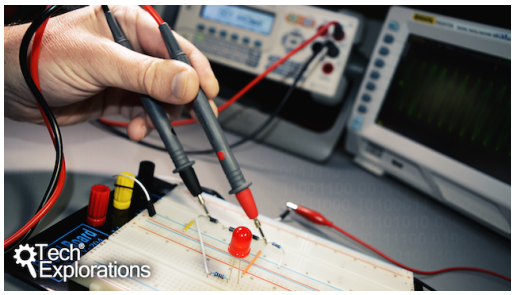
- Use matrix or LED strips, display text, graphics, and animation.
- Create motion with various kinds of motors and controllers.
- Networking with Ethernet and Wifi. Communications with Bluetooth, Bluetooth Low Energy,
- Communications with highly-reliable packet-based radio, and simple, ultra low-cost radio for less critical applications
- Multiplying your Arduino's ability to control external devices with shift registers and port expanders.
- Much, much more (for a full list, please look at the course curriculum)

You can find more details at https://techexplorations.com/courses/arduino-step-by-step-getting-serious/

Basic Electronics for Arduino Makers

All your Arduino work will be much easier and enjoyable if you have a good understanding of basic electronics. This is why we created this course: Basic Electronics for Arduino Makers.

I have designed this course for anyone with a basic understanding of electronics, who has already spent time tinkering with Arduinos.

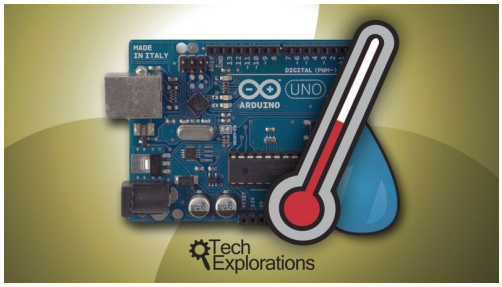


By the end of this course, you will have learned how to use commonly used components found in Arduino projects. You will also have learned how to do the relevant measurements and calculations to help you select appropriate components for your projects.

To complete this course, you will need a few cheap and common components and tools: resistors, capacitors, transistors, LED, diodes, and batteries. You will also need a multimeter, a small breadboard and jumper wires. All of these are probably things that you already have.

You can find more details at https://techexplorations.com/courses/basic-electronics-for-arduino-makers/

Arduino: Make an IoT environment monitor

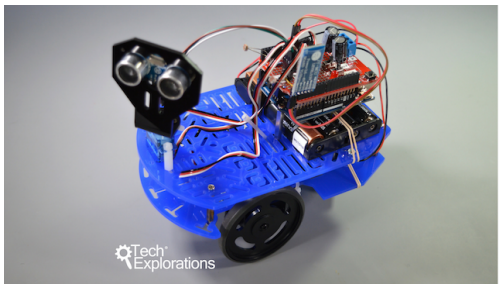


Project-based learning is the best kind of learning, especially when it comes to technology. At Tech Explorations, we would not be true to our mission if we didn't have interesting and fun projects that will challenge and stretch your knowledge. A perfect first project is "Arduino: Make an IoT environment monitor".

In this project, you will construct an environment monitoring system. You will put together piece by piece, and line by line. It will measure the environment conditions in your lab, and send them to the Internet where you will be able to access the data in a rich graphical dashboard.

You can find more details at https://techexplorations.com/courses/beginning-arduino-make-a-environment-monitor-system/

MAKE AN ARDUINO ROBOT



For a more challenging project, consider "Make and Arduino Robot". We have designed this Arduino project course as an opportunity for you to get deep into the messy details of understanding, designing and constructing a simple yet infinitely extensible wheeled robot.

From figuring out what the robot is supposed to do, to selecting the right parts, configuring them, assembling them and testing them, all the way to producing a refined outcome, this project emphasises the iterative process of problem-solving.

By the end of the course, you will have created an Arduino wheeled robot that can navigate towards a light source and avoid obstacles along the way, on its own. But more than that, by the end of the course you will have become a better problem solver. You will have experience in analysing problems and designing solutions. You will be able to integrate simple mechanical parts with motors, batteries, sensors and other electronics.

You can find more details at https://techexplorations.com/courses/make-an-arduino-robot/

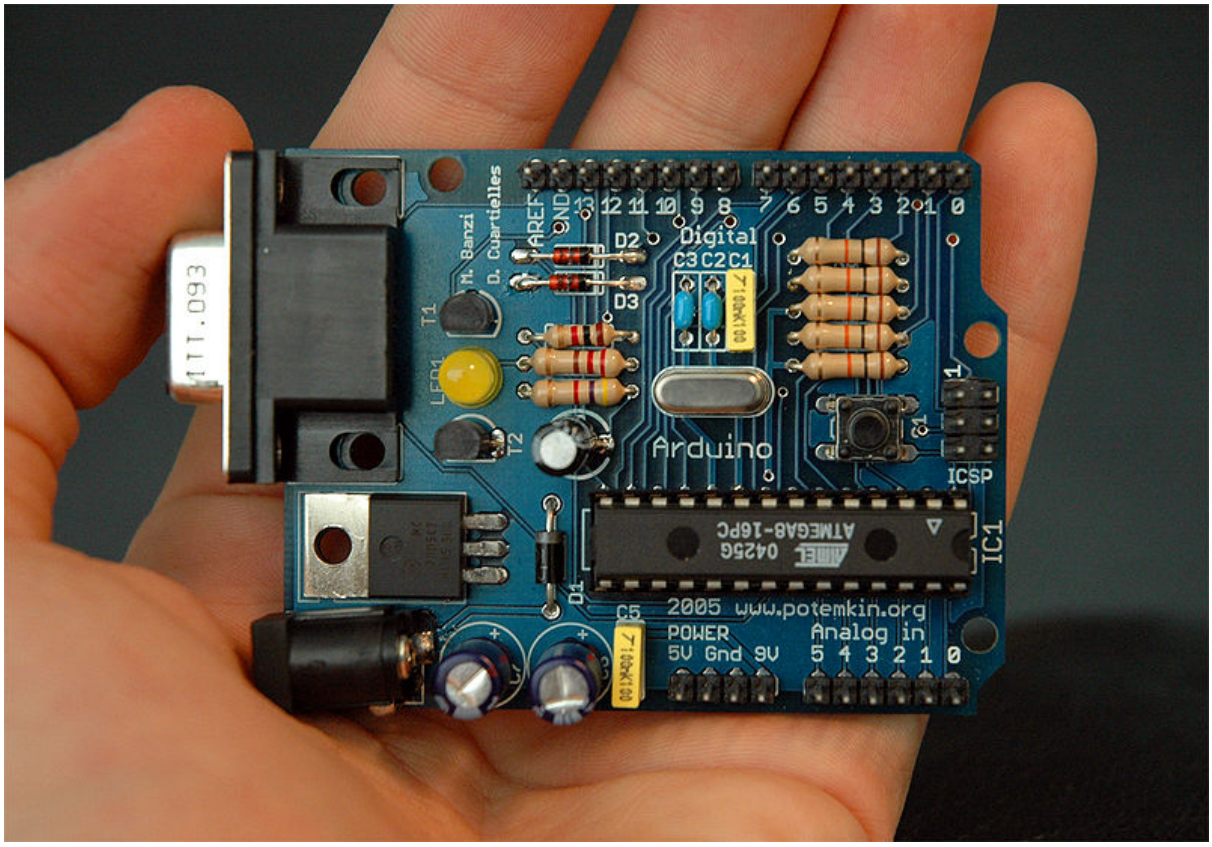
WITH ALL THIS SAID AND DONE, LET'S BEGIN WITH THE FIRST LESSON IN THIS BOOK…

WHAT IS THE ARDUINO?

Let's start from the very beginning. What is the Arduino, and where did it come from?

The Arduino is not a single thing, but a prototyping platform. The platform is a collection of hardware, software, workflows and support networks 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 the processor in your computer, except that it is very cheap, much "weaker" in terms of performance, and it is 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 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 own 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.

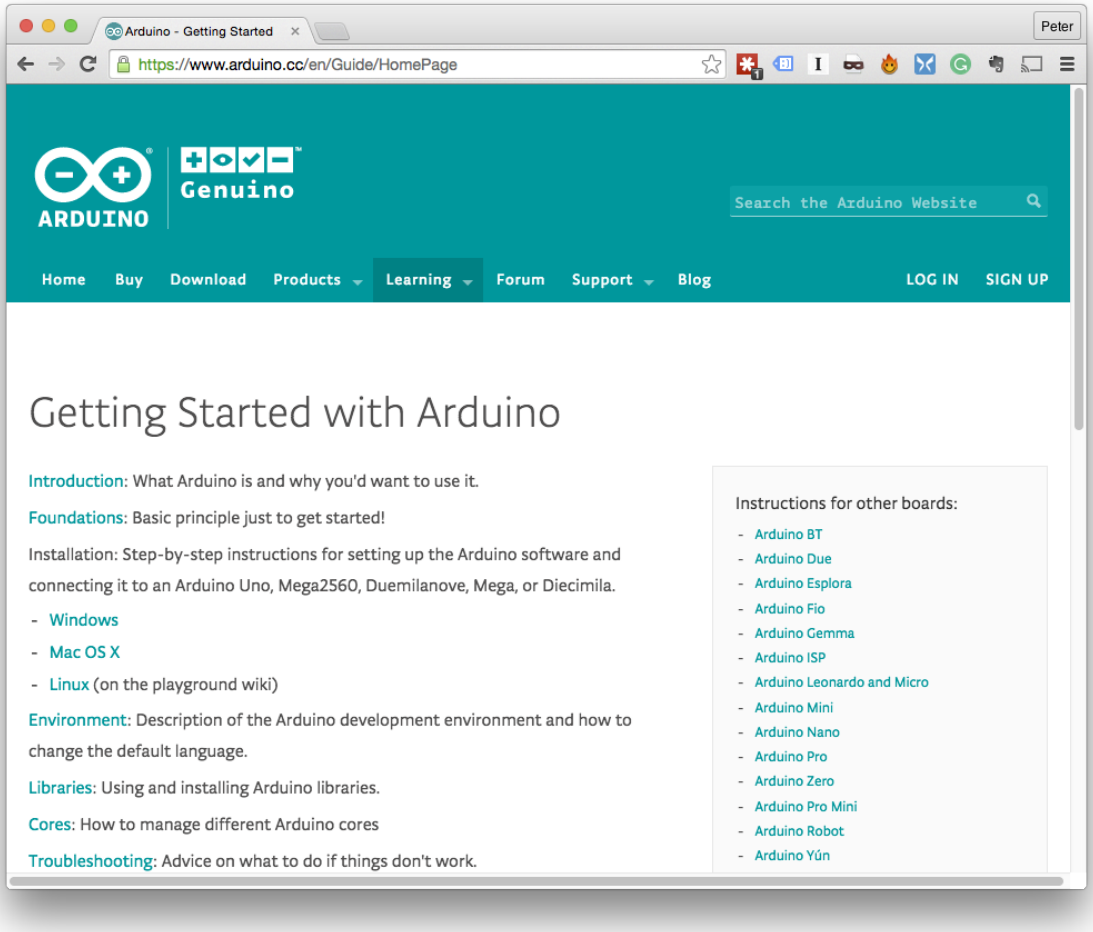
Wiring is what we now call the "Arduino Language" and the Integrated Development Environment (IDE), the core components that is a bit like what the HTML and the first graphical web browser were for the Web: It made the Arduino platform easy to use so that people who are not engineers can build sophisticated microcontroller-based gadgets.

Places to find help

As you go about exploring the Arduino, you will hit roadblocks. These are great opportunities for learning! The Arduino is very well documented, with a lot of places where you can find documentation and personalised help from other Arduino enthusiasts. Here are a few places worth visiting:

Arduino.cc

This is the premier resource for anything Arduino. This is the home of the original documentation for the IDE, bundled libraries (more about this is coming up), and the hardware. To access this documentation, go to https://www.arduino.cc/en/Guide/HomePage.



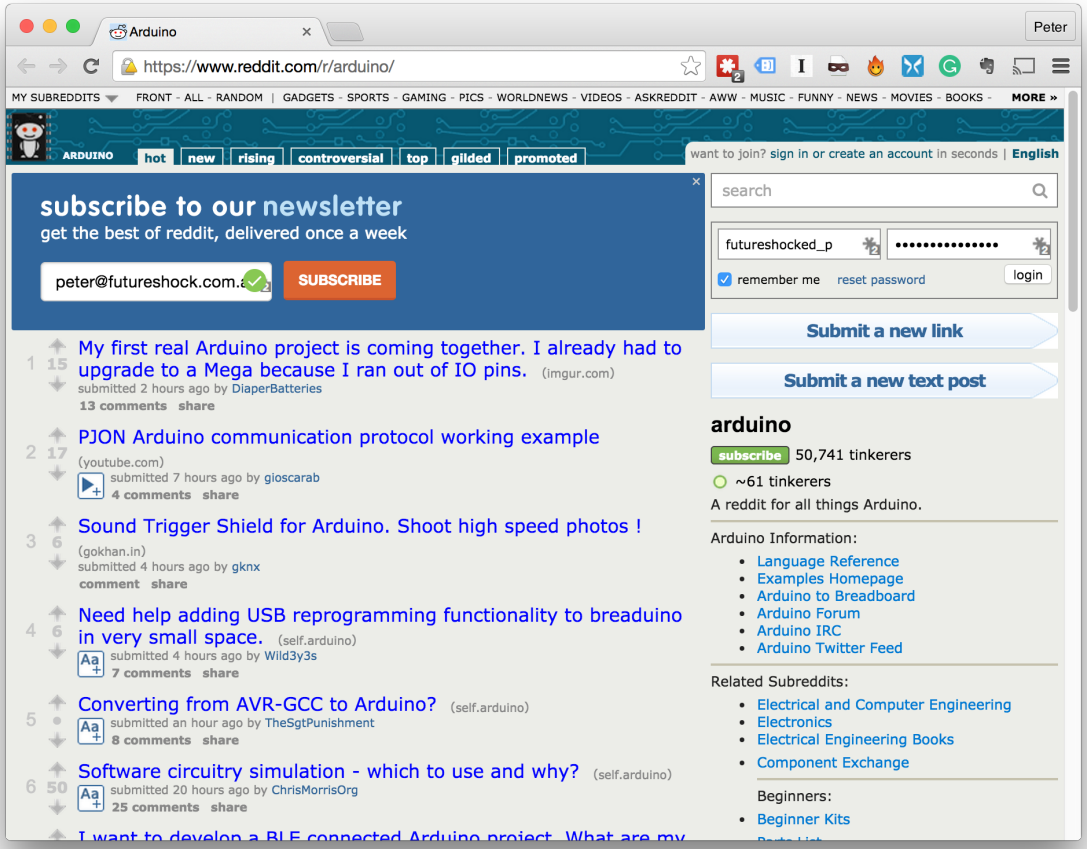
The home of the Arduino.cc documentation resources.

There are also numerous discussion forums from where you can find or offer help. There are forums on general electronics, LEDs, microcontrollers, audio, sensors, robotics and many more. If you are an artist, you will find peers discussing relevant issues in the Interactive Art forum. If you are a teacher, you will find the Education and Teaching forum relevant. There are also forums geared for people based on their location and language, device, products, and more.

The forums home page is at http://forum.arduino.cc/.

REDDIT.COM

Reddit is an online community with people talking about virtually everything and anything. Each topic is called a "subreddit", and Arduino has its own.

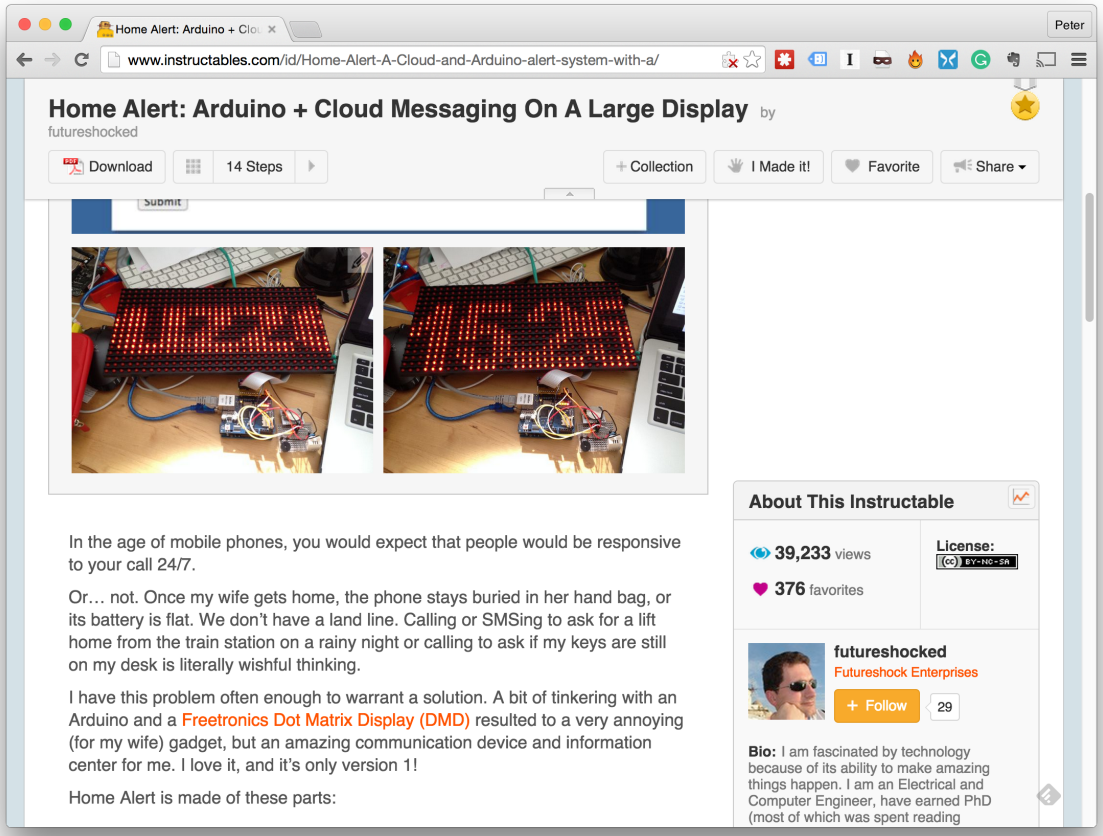


The Arduino subreddit

The Arduino subreddit is at https://www.reddit.com/r/arduino/. Here, people talk about their projects, ask questions, and participate in conversations.

INSTRUCTABLES

Instructables is a web site where people post project guides so that other people can go ahead and build their own versions of these projects.



Instructables

Instructables is a great resource because it contains details of things people have created in a way that other people can follow. If you want to learn how to build an Arduino-based cloud-powered home notification system, an Arduino stopwatch, or Arduino-powered musical chairs, then go to instructables.com

DANGEROUS PROTOTYPES

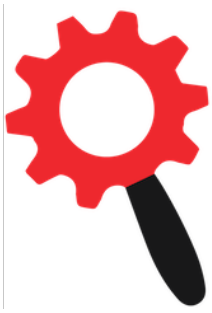
Dangerous Prototypes is a blog featuring relatively advanced projects. New projects are added every month, and they are mostly geared towards more advanced makers. These projects tend to push the limits of what you can do with microcontrollers, so they are worth looking at even if you are not yet at the skill level required to actually build them.

Dangerous Prototypes

If you want to know how to re-purpose your old Nokia 3100 as an Arduino shield so that you can send out SMS messages, this is the place to look at.

You can access the Dangerous Prototypes site at http://dangerousprototypes.com/.

TECH EXPLORATIONS



As a student of our courses, we provide support in two ways: Direct, one-on-one assistance via our Help Desk, and community support via our course forums.

Our Help Desk is where you can ask for help for any technical website issues, billing issues, or problems that came up using your study. We typically respond within 24 hours during work days.

Each course has a dedicated Forum. This is the place where all students and instructors discuss course-related issues, exchange ideas, and help each other. Are you having trouble understanding something in a lecture? Is your Arduino behaving badly? Are you having trouble with a sketch? The Forum is where you want to ask your question and tap on our Community's collective expertise.

NOTABLE VENDOR WEB SITES

The major electronics hardware vendors often also provide excellent documentation and guides for the products they sell.

In particular:

- Element14 has their Arduino guides and forums at http://www.element14.com/community/groups/arduino
- Adafruit at https://learn.adafruit.com/category/learn-arduino,
- Sparkfun at https://learn.sparkfun.com
- SeeedStudio, at http://wiki.seeedstudio.com/

These resources are free to use regardless of whether you have purchased one of their products.

ARDUINO BOARDS

Coming across an Arduino board for the first time can be intimidating. Only a few years ago, there were just two or three boards to choose from. Today, there are dozens of boards designed by Arduino and various manufacturers. Some, but not all of these manufacturers, work closely with Arduino to ensure that the boards they produce are fully compatible with the official Arduino boards. The boards produced by these manufacturers are stamped as "official" and are listed on the Arduino.cc web site products page.

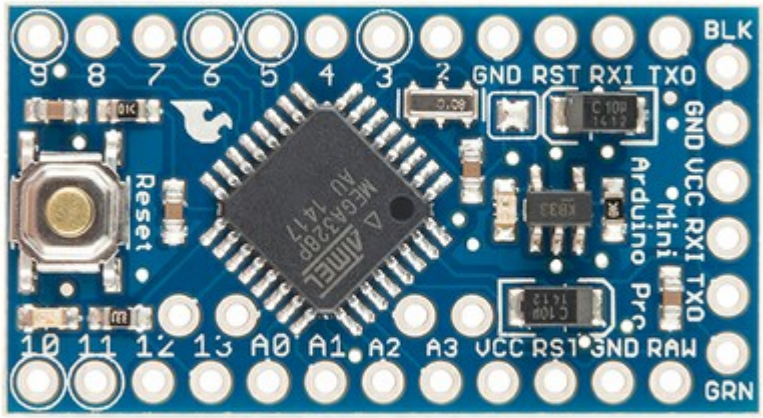
If a board is not listed on the Arduino products page, then it probably a clone. In my experience, clones will work like the original, but usually they are made with cheaper components that will wear off after a while, especially the connectors in the headers. It is worth investing in official Arduino 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.

At the time I am writing this, there are around 20 official Arduino boards.

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 (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.

The official Arduino Uno. The best board for the beginner and beyond.

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.

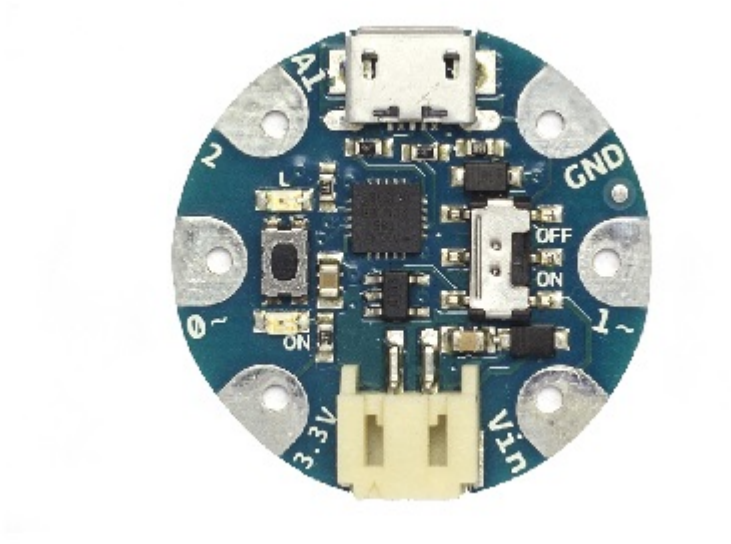


The tiny Arduino Pro Mini

There are also Arduinos based on more capable microcontrollers, like the Mega, or even microprocessors running Linux, like the Arduino Yún, which also has built-in Wifi capability and is well suited to Internet of Things applications.

The Arduino Yún, geared for Internet of Things applications

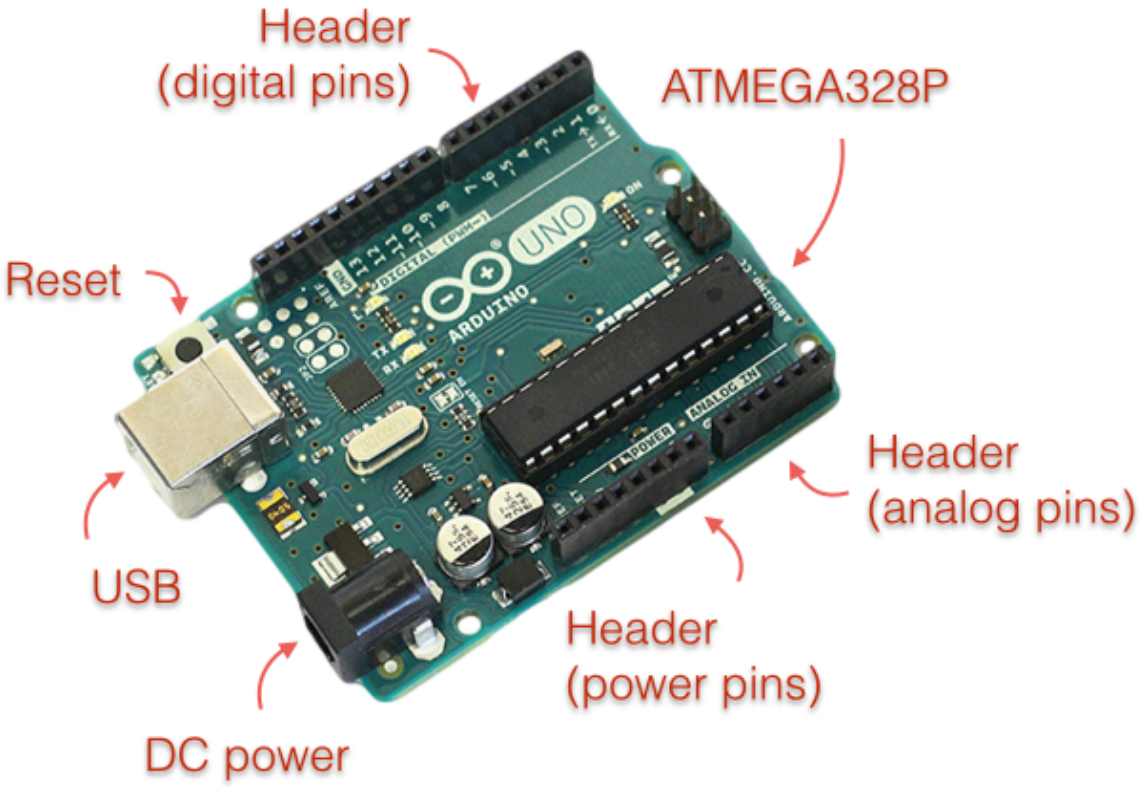
Worth mentioning are also the various wearable Arduinos. If you are interested in making electronics that you can embed in clothing, then you can look at something like the super-tiny Arduinos Gemma or Lillypad. These are small, battery-ready, low power Arduinos, circular in shape so that they don't catch on fabrics.



The super-tiny Arduino Gemma

PARTS OF AN ARDUINO BOARD

Arduino boards have certain features that are shared between them and are good knowing about. I'm describing the most important ones in this image:



The most important parts of the Arduino Uno

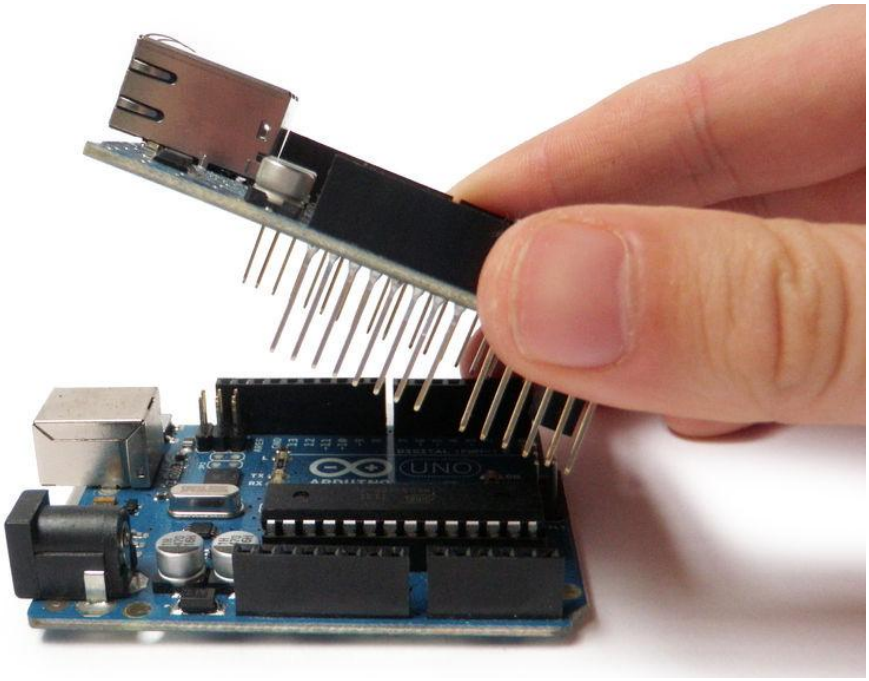
COMPONENTS

The Arduino can't do much on its own. Its purpose is to control components that you connect to it. 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.

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.

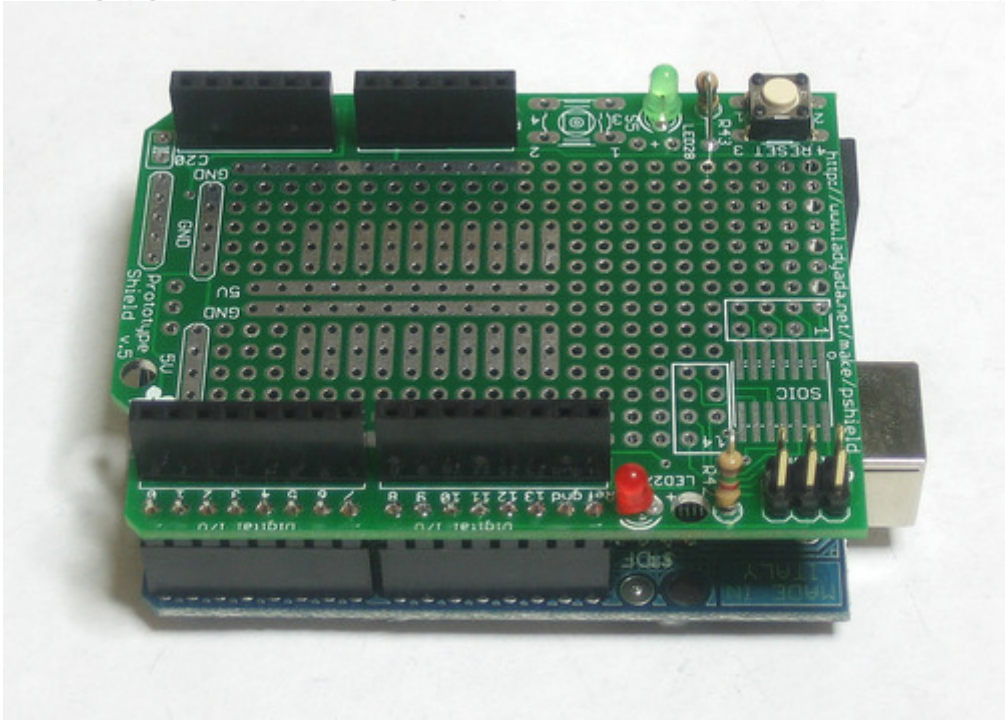
Most shields are built to work with the Arduino Uno, and as a result virtually all other full-sized Arduinos have an Uno-compatible header configuration.



The Arduino Ethernet shield (top) about to connect to an Arduino Uno (bottom). To make the connection, simply 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, datalogging, sensors, color LCD screens, and more.

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



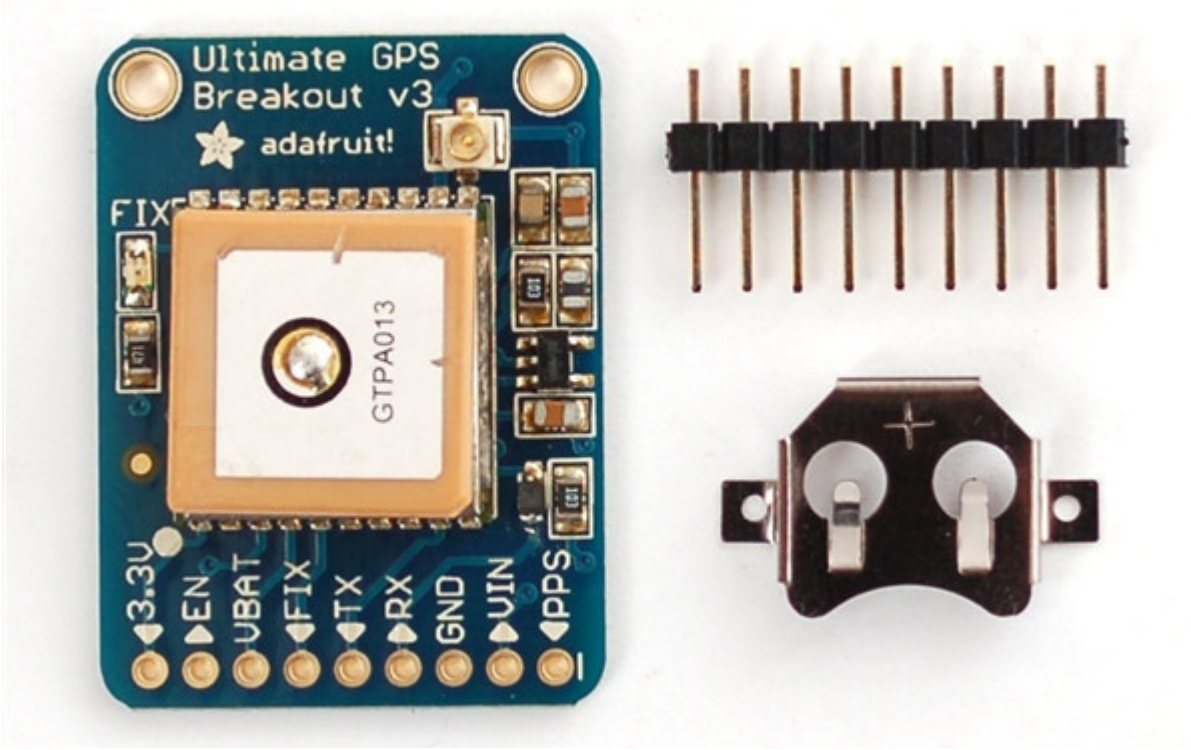
A prototyping shield like this one from Adafruit makes it easy to preserve your best circuit designs.

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

Breakouts

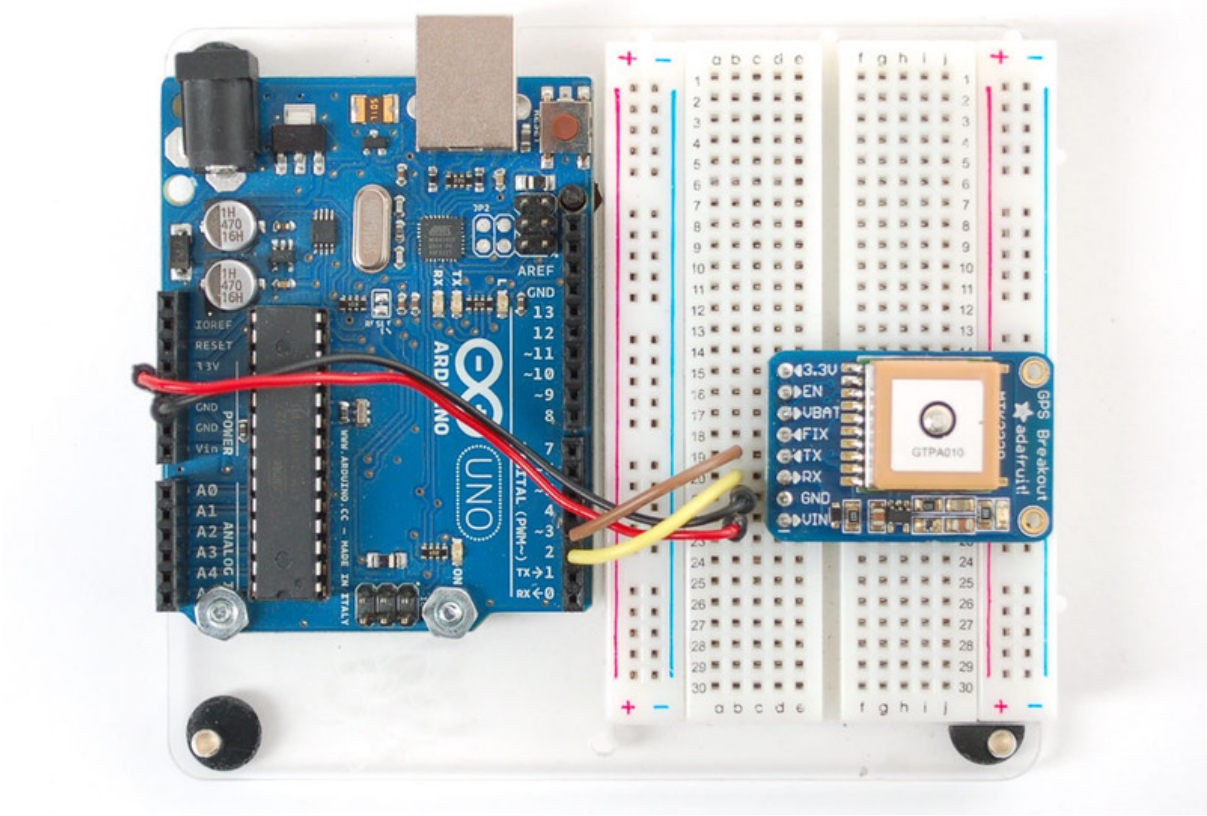
Breakouts are typically small circuit boards built around an integrated circuit that provides a specific functionality. The board contains supporting circuitry, like a subsystem for providing power, LEDs for indicating status, resistors and capacitors for regulating signals, and pads or pins for connecting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

In many cases, the same functionality is offered in a shield or a breakout format. For example, you can get the exact same GPS system as a breakout or as a shield. In such cases, the difference is size. The breakout is smaller, and 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)

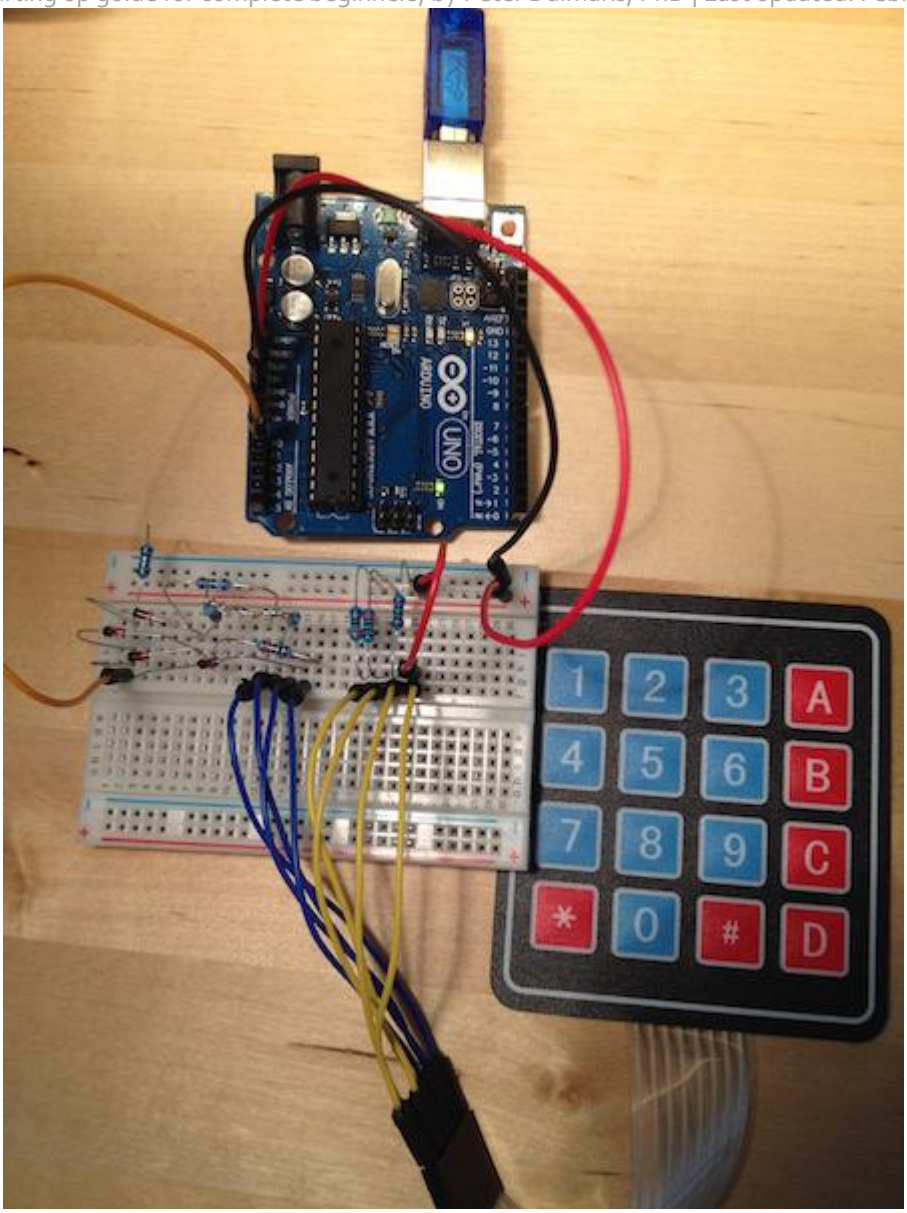
Some times, apart from using jumper wires to connect the breakout to the Arduino, you may also need to do a bit of soldering, like I had to do for the GPS breakout. Here's the quick version of how this soldering job went (video on Youtube, txplo.re/asupe1574).

The really nice 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 good for learning, breakouts can be embedded into a final product.

Components

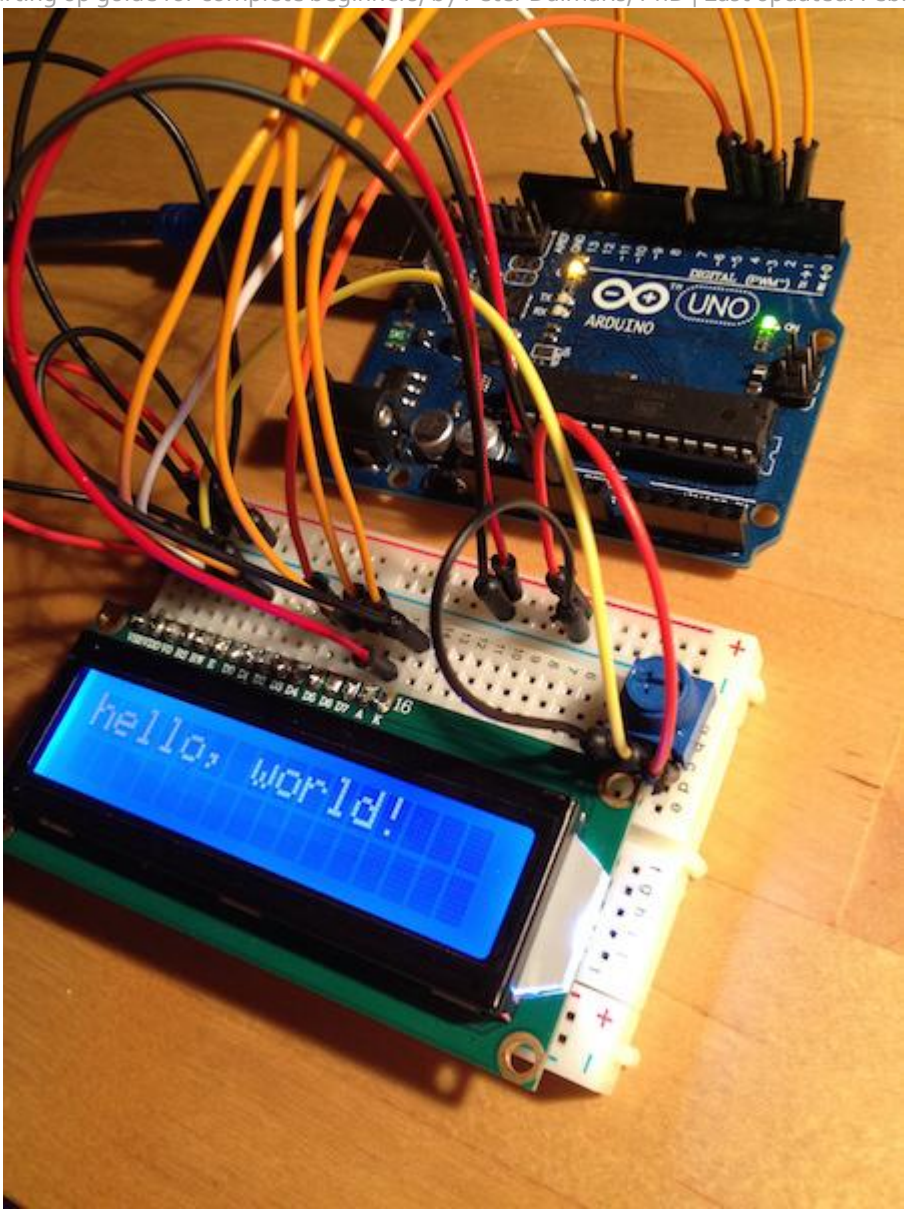
While breakouts give you easy access to components by putting them on a board with their supporting electronics, you will eventually need access to the individual component so that you can fully customize the way it works in your circuit.

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 properly, you will need to add several wires and resistors.

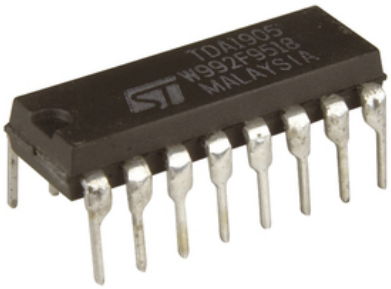


Using a 4x4 keypad requires external wires, diodes and resistors. This is more work (compared to a shield) but often the flexibility you get in return is worth the effort.

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 thumb print scanners can be connected to an Arduino as components.

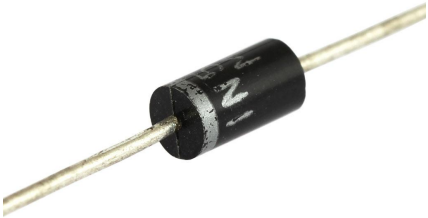
Discrete components

At the bottom of the scale in terms of size and complexity we have a wide range of discreet components. Things like resistors, capacitors, transistors, LEDs, relays, coils etc. fall under 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 electrical 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 than the examples mentioned.

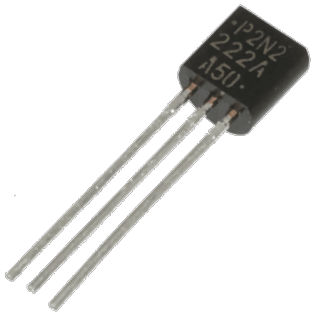
A resistor limits the amount of current that flows through a wire

A capacitor stores energy, or works as a filter

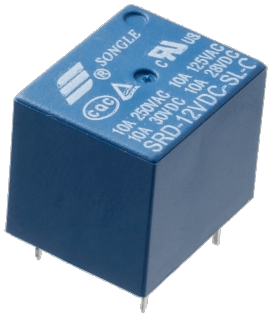


A diode limits current to flow towards one direction only

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 start 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.

What's it like programming for the Arduino?

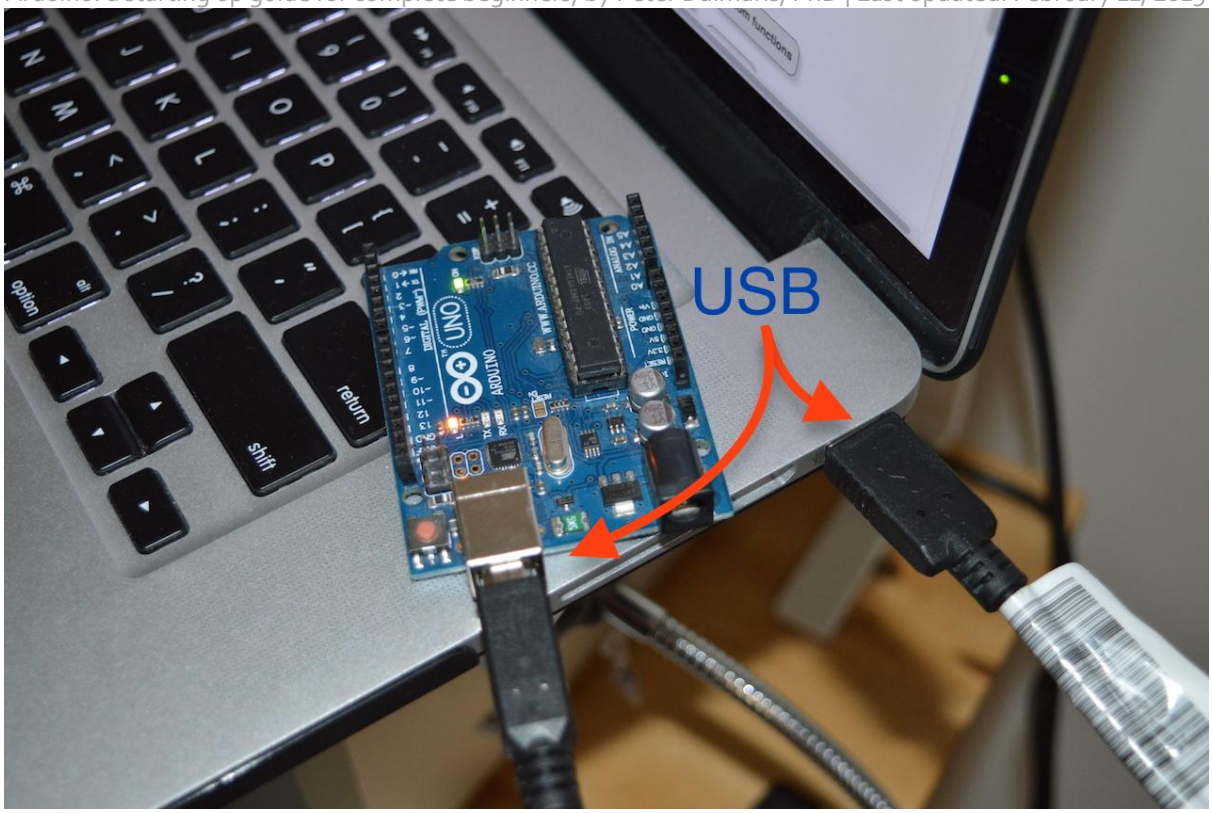
To write programs for your Arduino, you use a simple tool called the "Arduino IDE".

Advanced makers have the habit of replacing the one provided by Arduino with one of their choice in order to take advantage of various advanced features, but you don't have to worry about this until much later.

No matter what your platform is, Windows, Mac or Linux, the Arduino IDE looks like this:

The official Arduino IDE

Programming text editors don't come much simpler than this. You type the program into the white box, and as you type, the IDE will recognise keywords based on their type and highlight them with a special colour for each type. When you are ready to upload, connect your Arduino to your computer with a cable:



Connect your Arduino to your computer with a USB cable

In most cases, the IDE will detect the connected Arduino and will configure itself with the correct USB port. If not, you can use the Tools menu to set the Arduino board model and the USB port is connected to it. Here is a quick video on how to do this (Youtube, txplo.re/ardui32cc).

A few seconds later, the upload will finish and your sketch will start running on your Arduino. In this example, I uploaded a sketch that simply makes one of the LEDs on the Arduino board itself to blink. And here is what that looks like (Youtube, txplo.re/ardui383b).

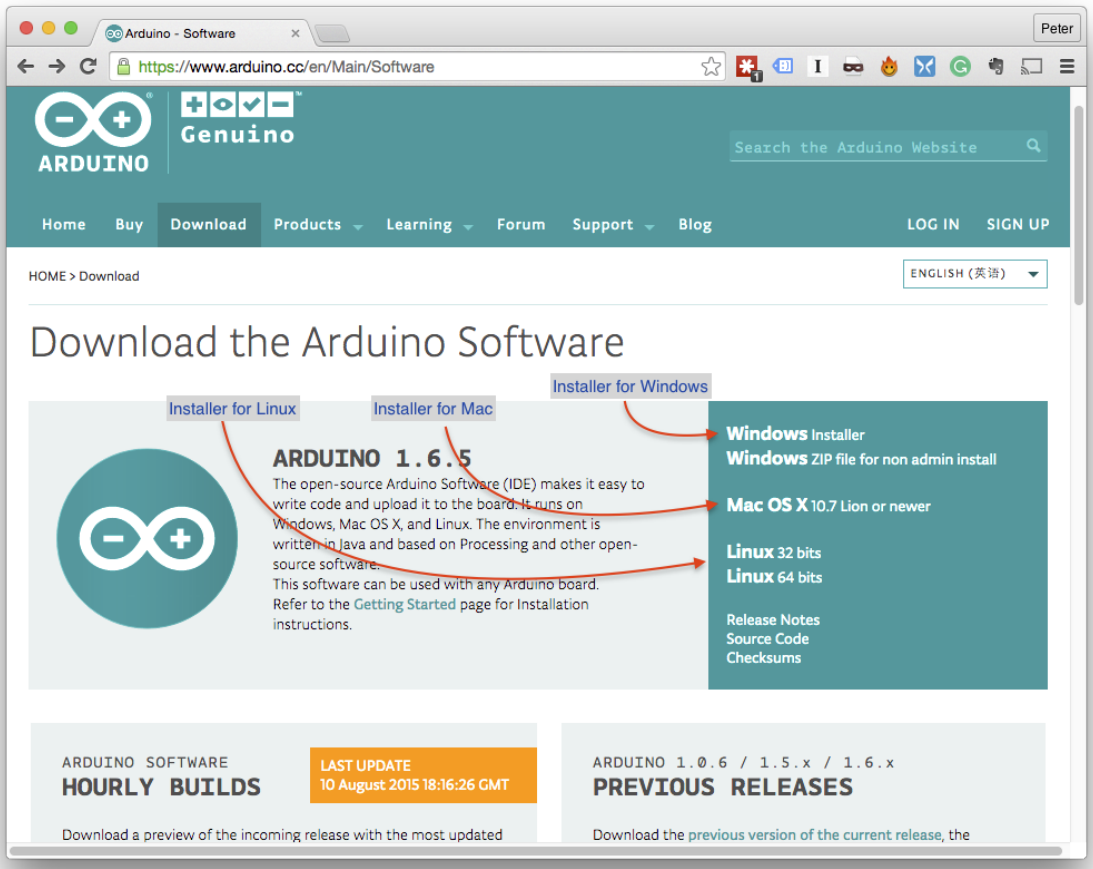
And this is what the process of connecting and uploading a sketch to your Arduino looks like.

The Arduino engineering team has done an amazing job at making a once complicated process almost as easy as sending an email!

QUICK SETUP GUIDE

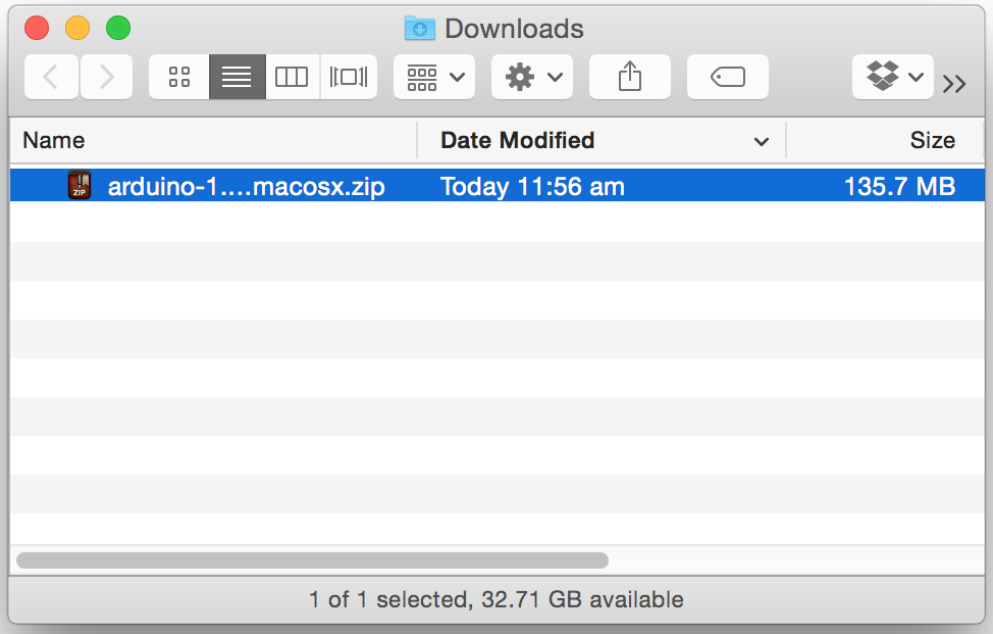
Now that you know what to expect, let's go ahead and setup your IDE. The process involves downloading the software from arduino.cc, and installing it on your computer. The only requirement is that you already have a Java runtime environment already installed. This is usually not a problem on Windows and Mac computers.

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

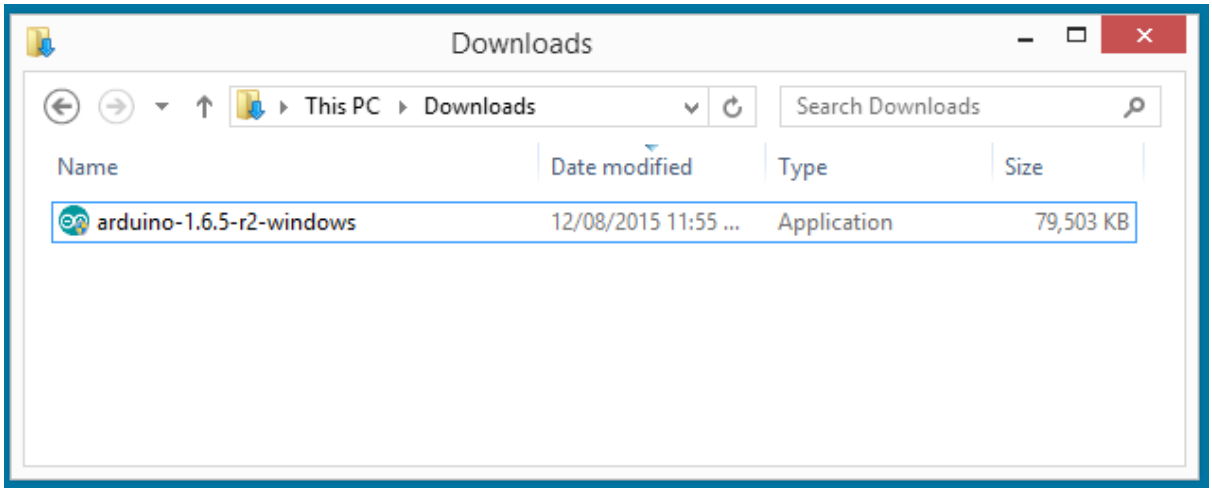


The Arduino IDE download page. Pick the installer to match your computer operating system.

Look for the installation file in your download folder, "~/Downloads" for the Mac and "Downloads" on Windows.



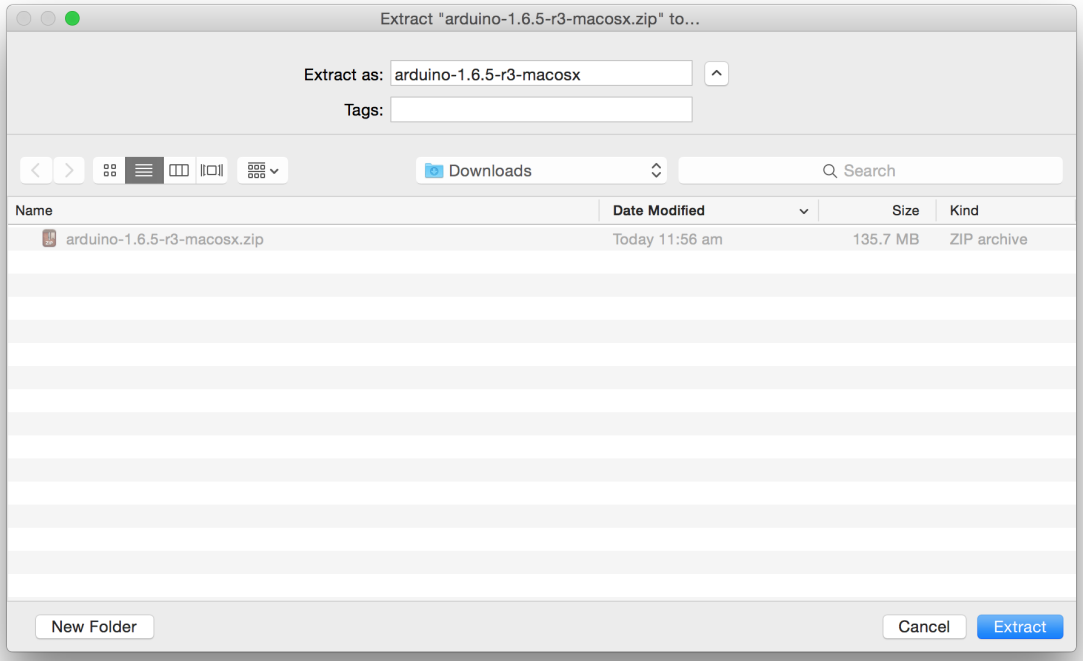
The IDE installer on the Mac



The IDE installer on Windows

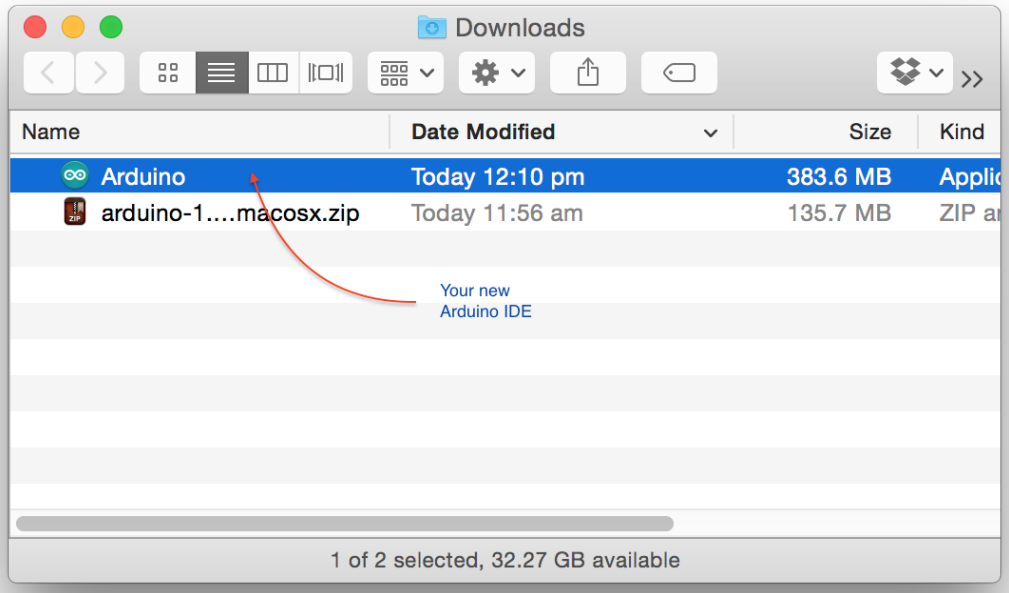
Installing on a Mac

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

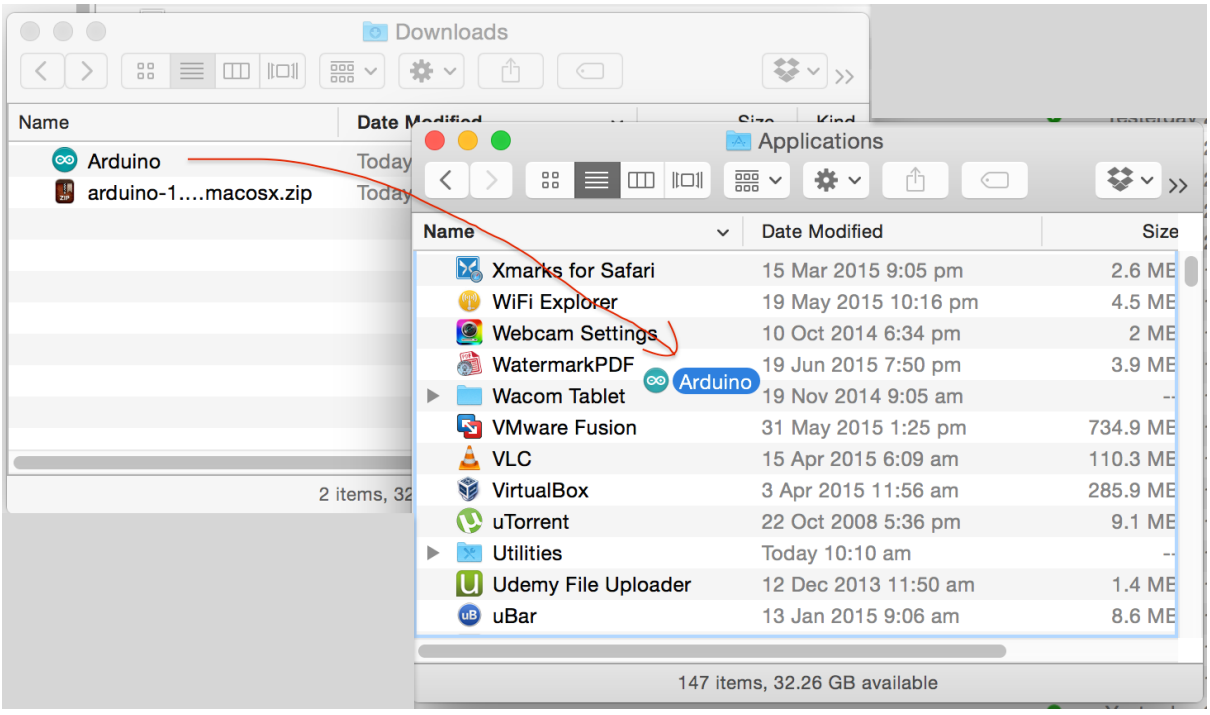


Extracting the IDE installer on the Mac

When the extraction completes, you will have 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.



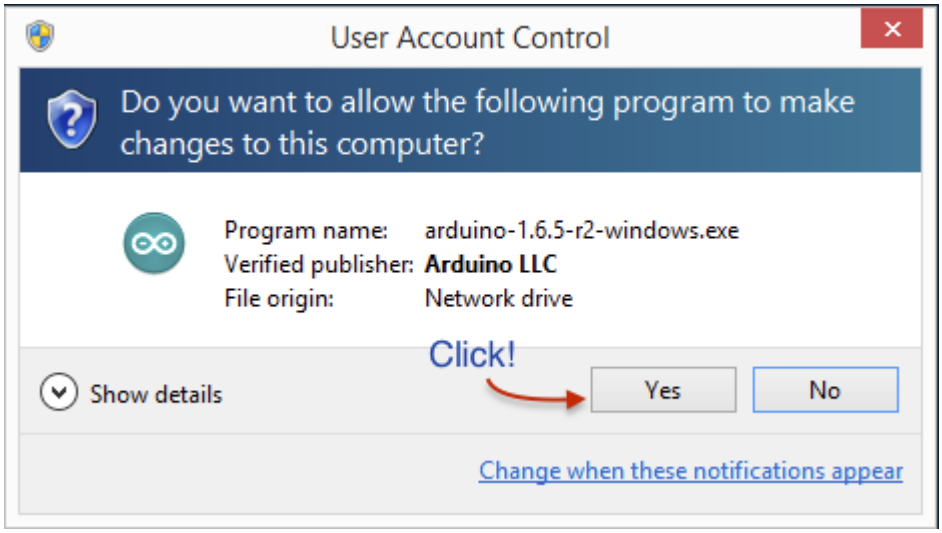
... And finish the process by copying the file into your Applications folder.

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

Installing 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:

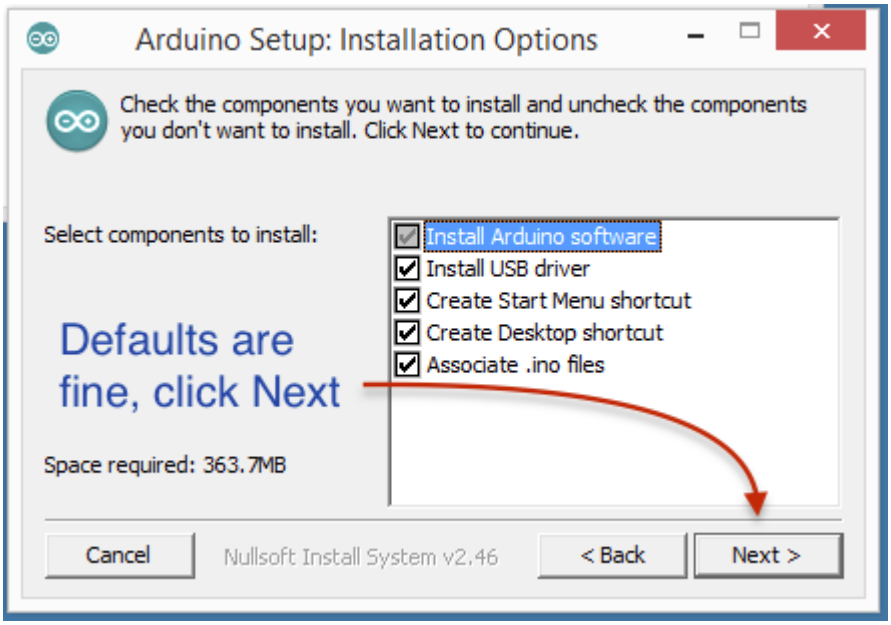


Yes! It is safe to continue!

Agree to the license agreement:

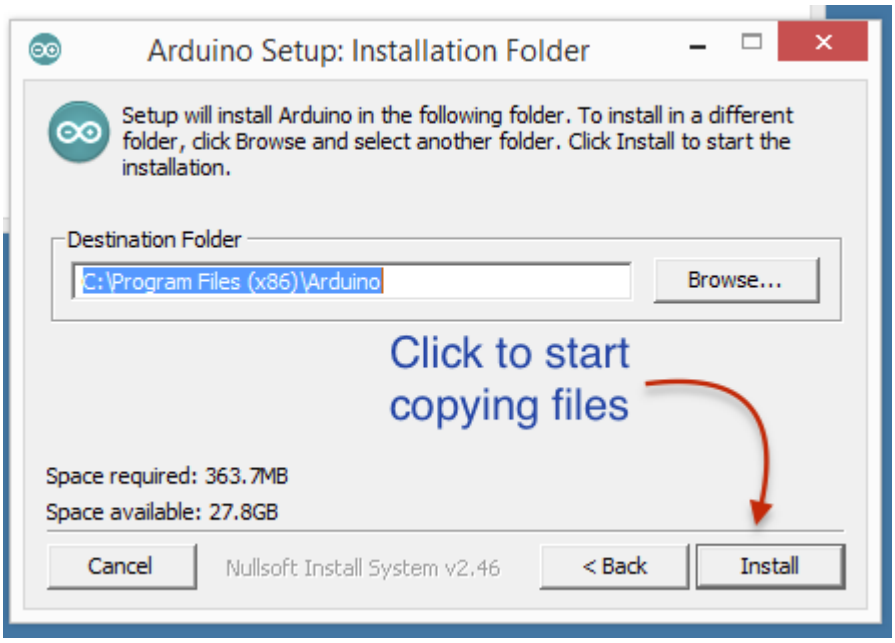
Yes! I agree!

Accept the checked components:



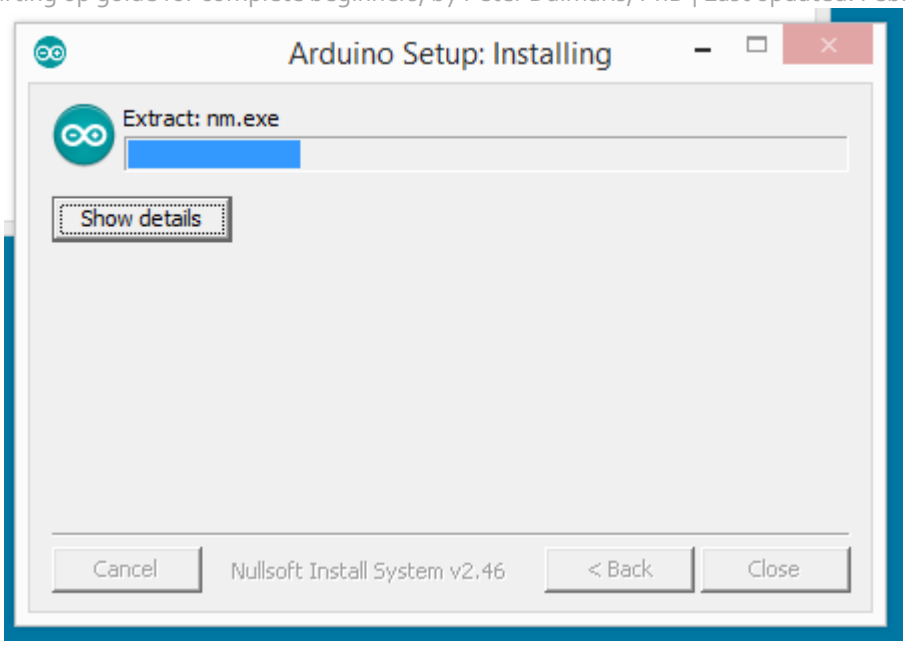
Yes! All these components are useful.

Accept the default installation location, and the file copy will begin:



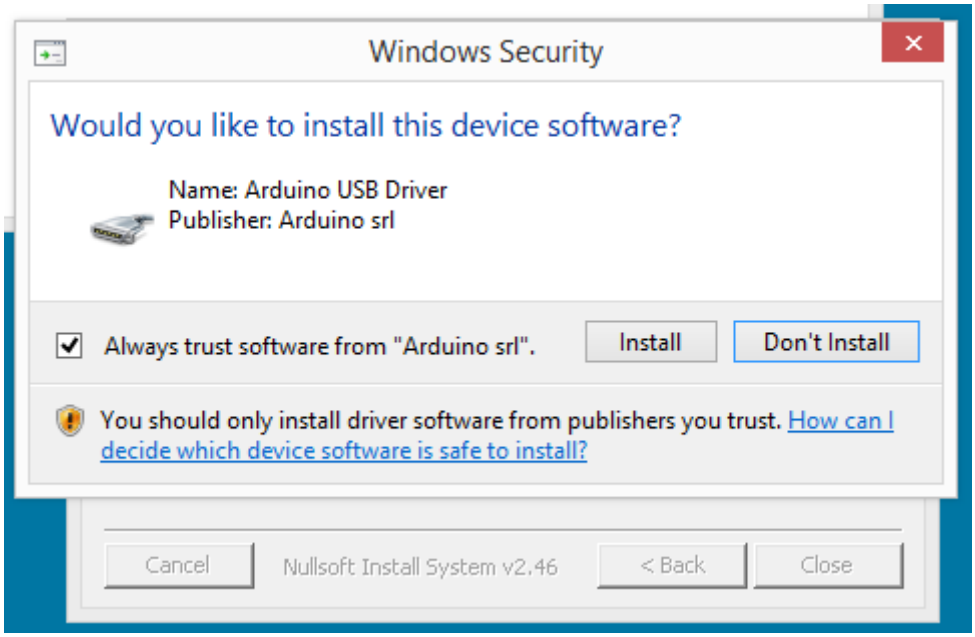
Yes! This location looks fine

The installer will start copying files to the installation location:



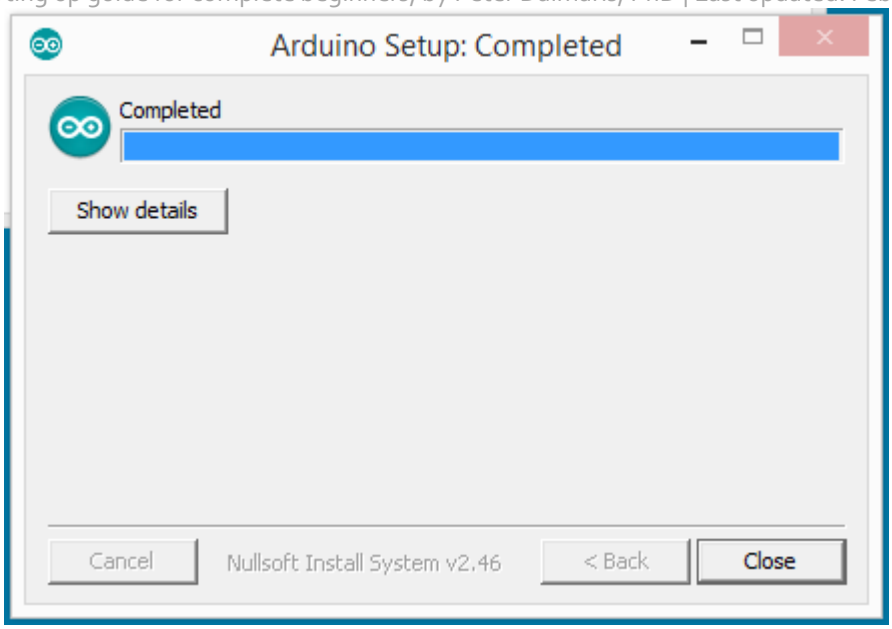
Copying files to the destination

You will be asked to install device drivers a couple of times. Just click on "Install":



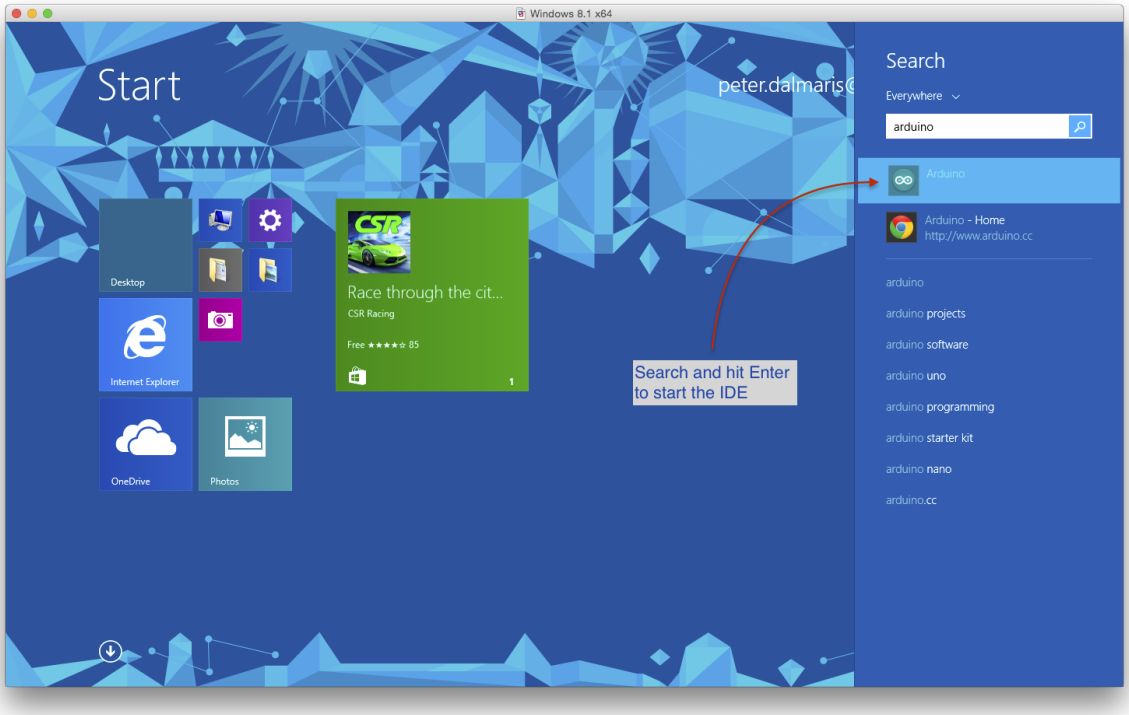
Yes, these USB device drivers are also useful

The copy process will finish. Click "Close" to close the installer:

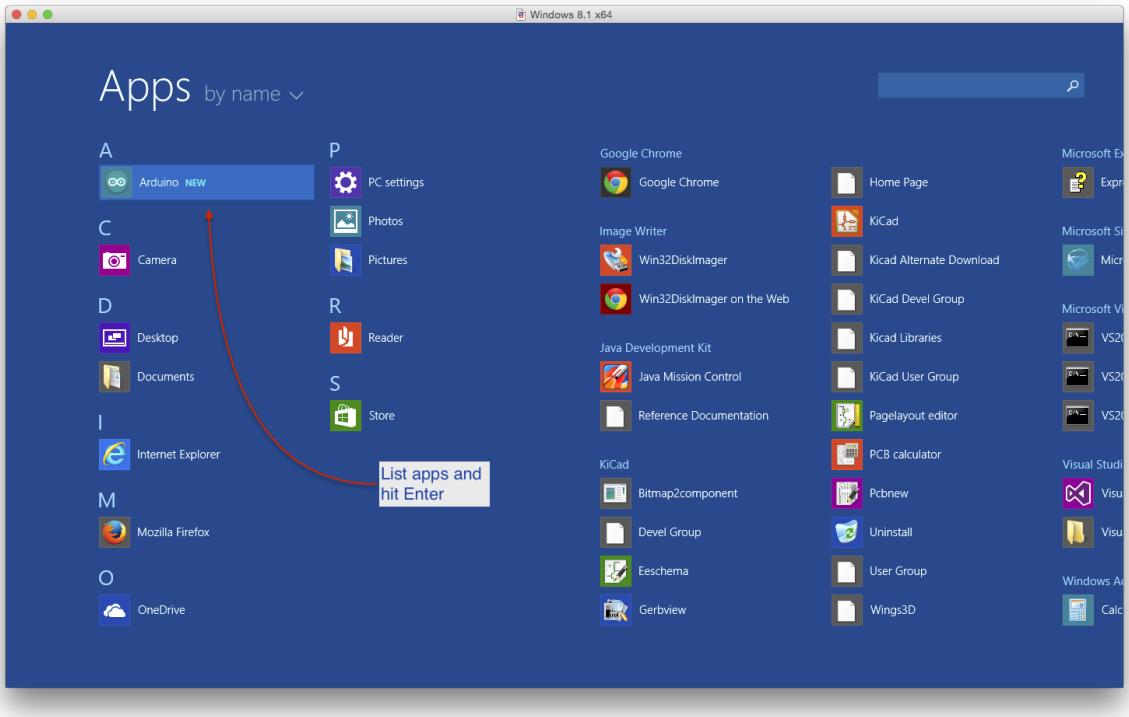


Ok, the installer is finished. Let's close it.

You can start the Arduino IDE just like any other Windows program. You can search for it or use the program list:



To start the Arduino IDE, you can search for it

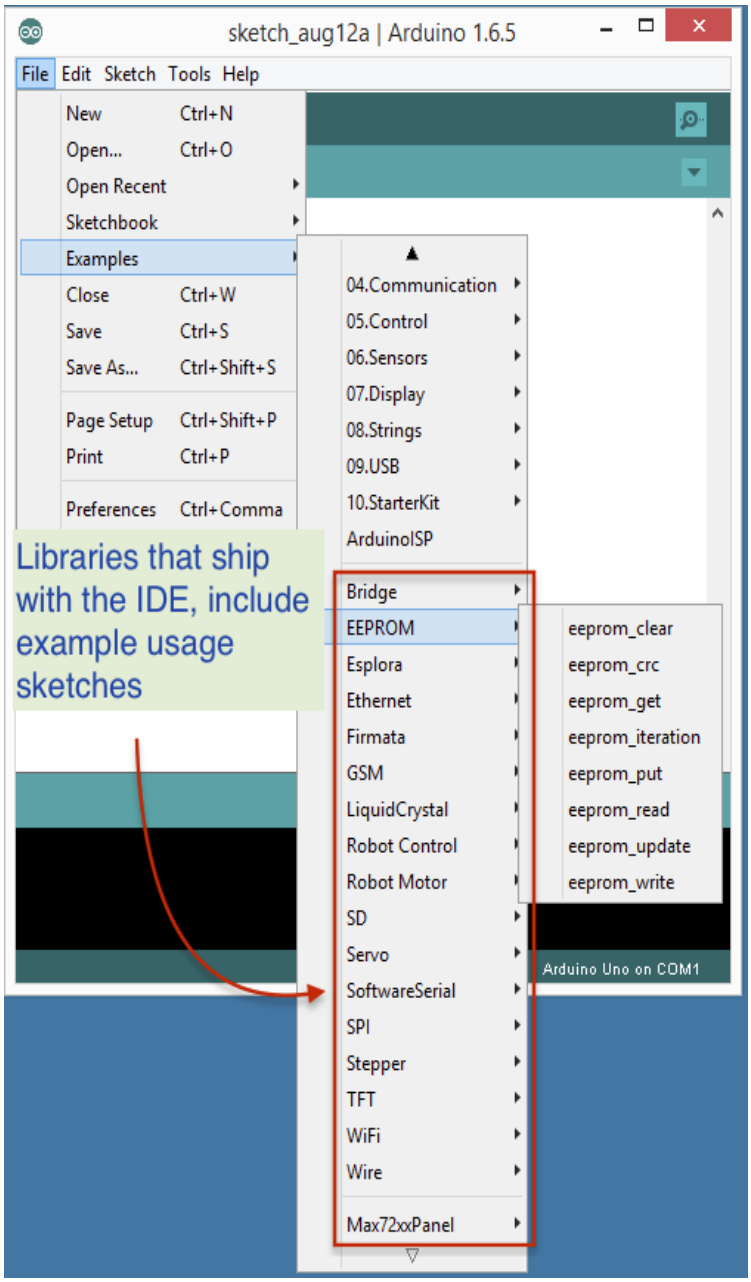


... or you can list all apps and look under "A"

ARDUINO LIBRARIES

A lot of software has already been written for the Arduino in a way that makes it easy for people to reuse in their own sketches. Often, this software is organised in a library and then distributed via the web, through sources like the arduino.cc site and Github, an online software source code repository.

Many useful libraries are included in the Arduino IDE by default. Have a look. Start the IDE, and then click on the File → Examples menu item (same for Windows and Mac). You will see a list of items, like in this screenshot:



A list of libraries that ship with the IDE

This is not a complete list of libraries, only those that have example sketches. The IDE contains many more libraries that are hidden in its installation directory.

Do a quick browse through them. You will find libraries that make it easy to connect your Arduino to an Ethernet network, to various peripherals through serial protocols like SPI, servo motors, Wifi networks, and to color graphics screens.

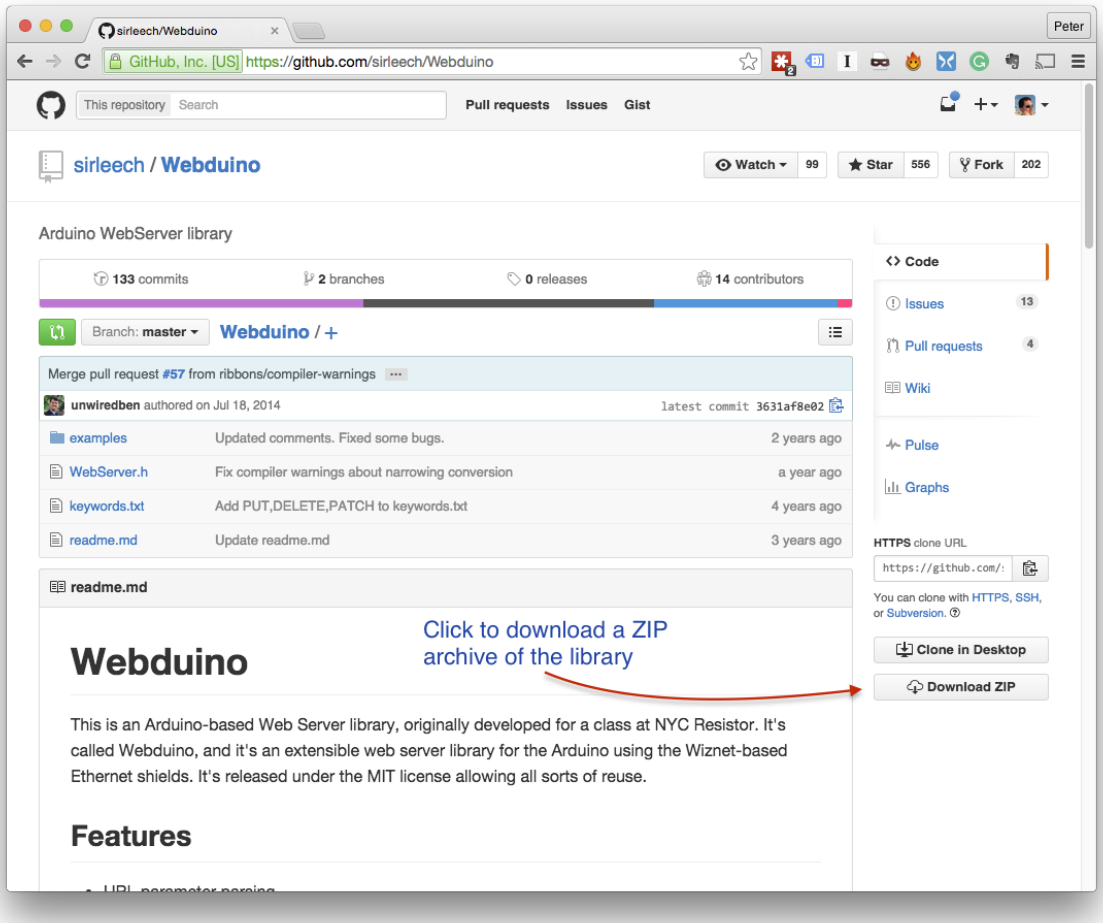
Installing a new library

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

Let's say that you want to have a small web server running on your Arduino. You can setup 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 just 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 library is available on Github at https://github.com/sirleech/Webduino.

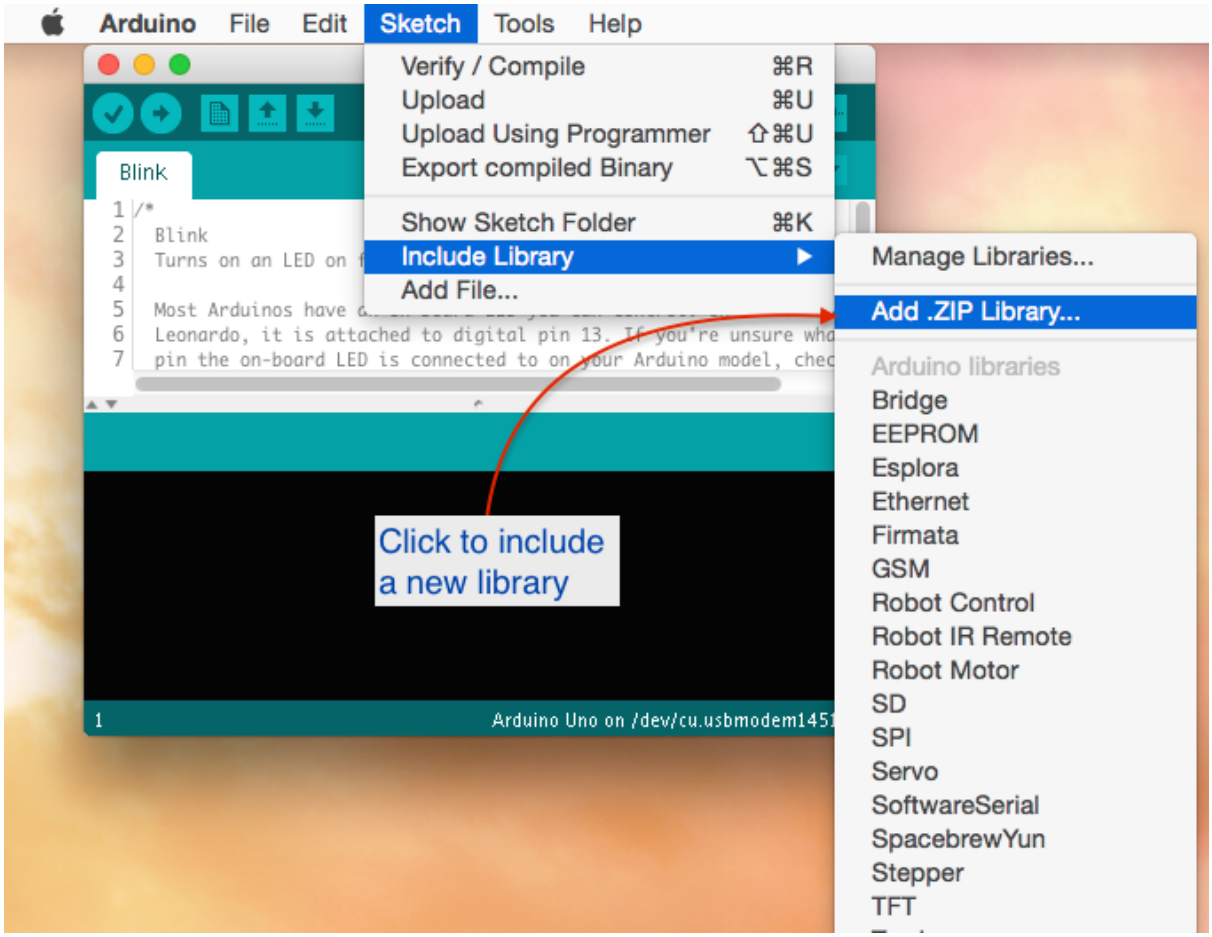


The home page for Webduino on Github

Download the Zip file on your computer. It doesn't matter what platform you are on, libraries work the same regardless of whether you are on Windows, Mac or Linux.

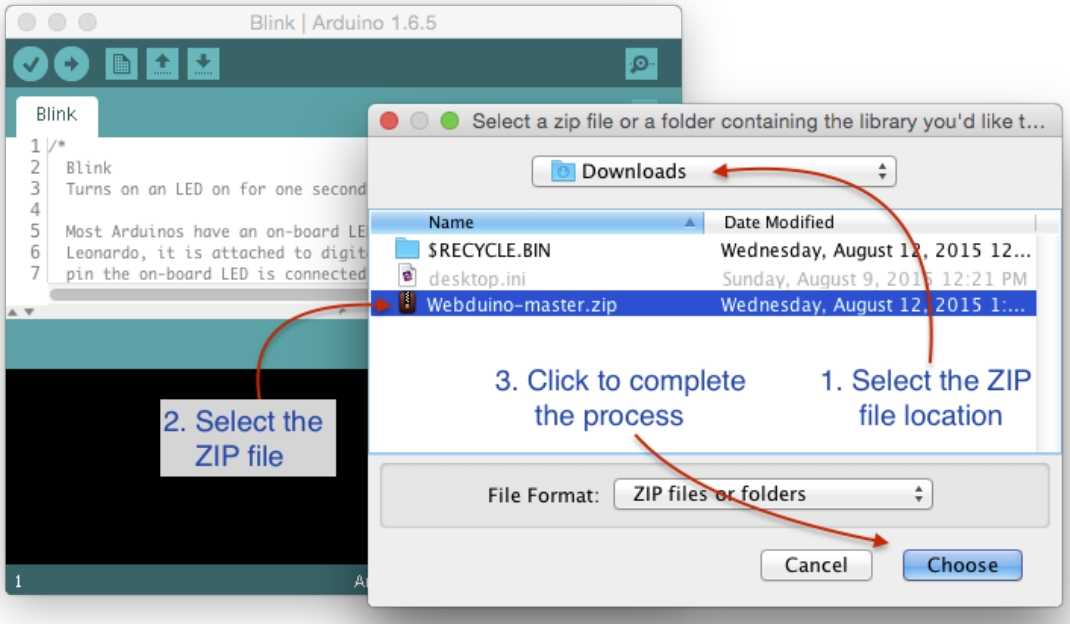
Also, don't worry about extracting the files from the ZIP archive. The newer versions of the Arduino IDE have an easy library installer that take care of extracting the library from the ZIP file and copying the files to the right location.

Assuming the library ZIP file is in your Downloads folder, start the Arduino IDE. Then click on "Sketch → Include Library → Add .ZIP Library…", like this:



Including a new library

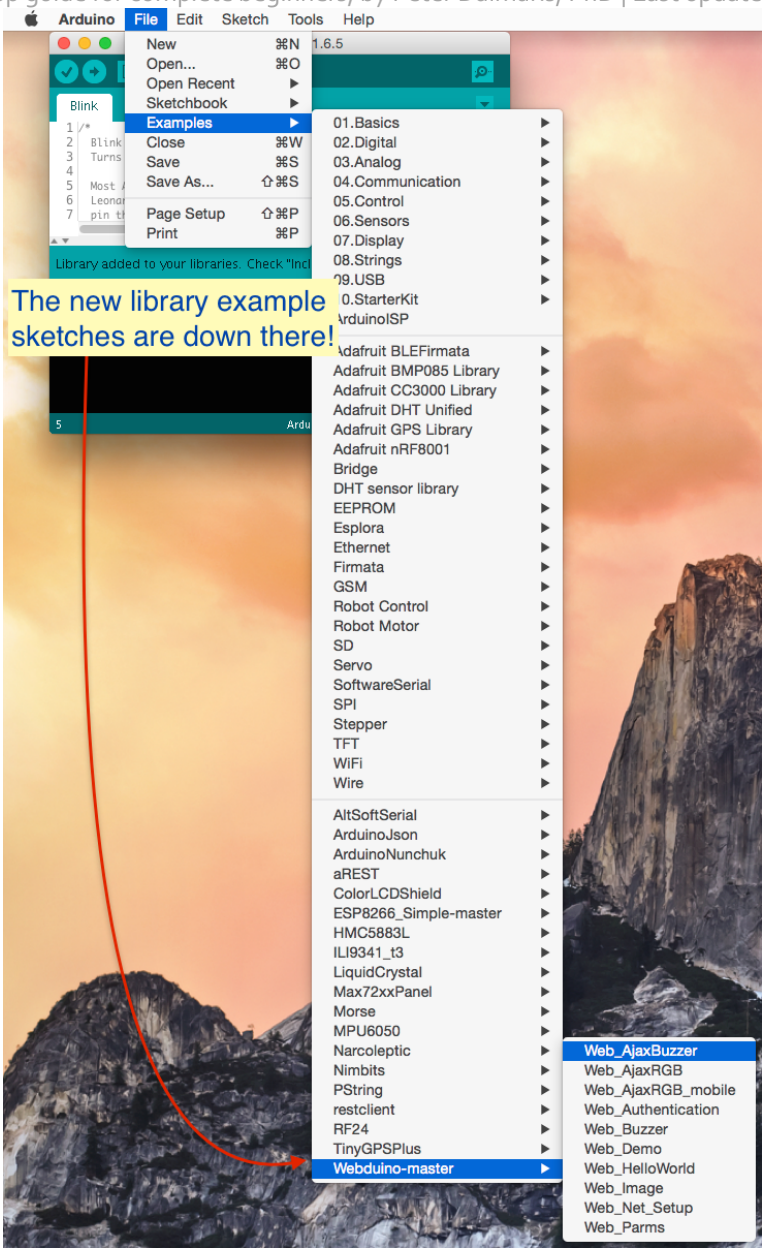
A new dialogue box will pop up. Browse to the location of the ZIP file, select it, and click on "Choose" to complete the process:



The Library addition dialogue box

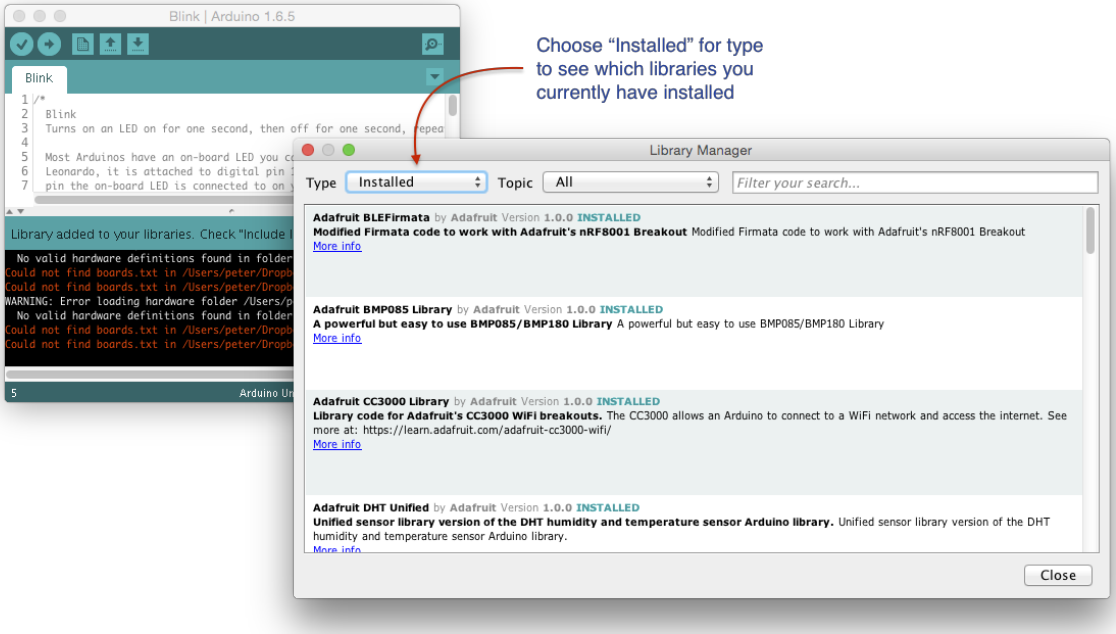
When you click on "Choose", the dialogue box will disappear, but nothing else is going to happen. No confirmation, no sound... To make sure that the Webuino library was actually installed, you can look for the example sketches that most libraries include.

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



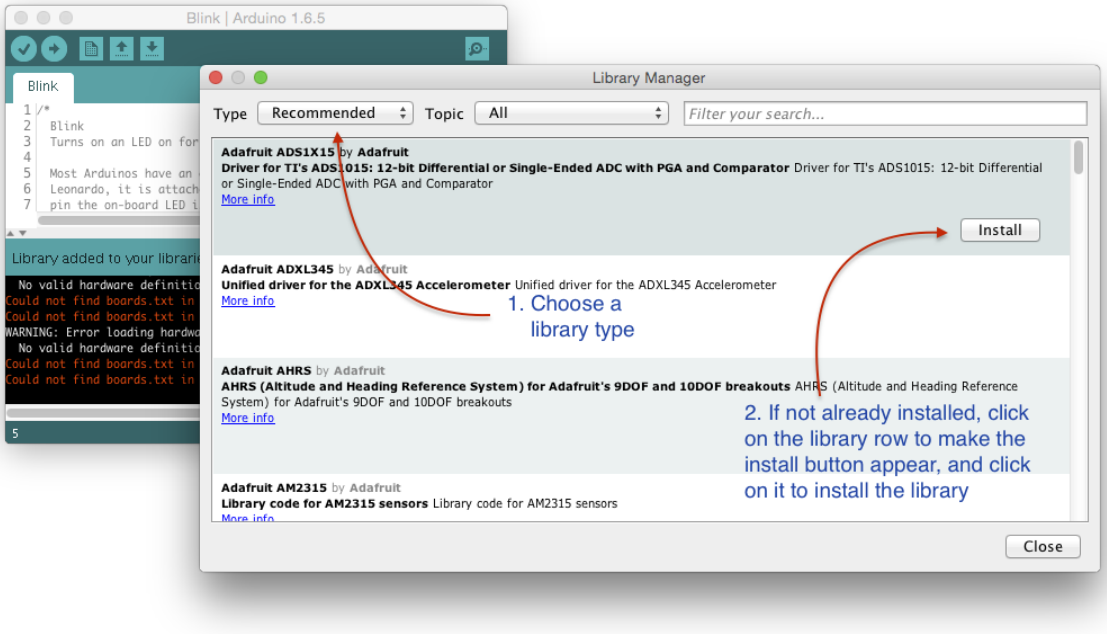
There's the new library, right at the bottom of the list!

You can also find a list with 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

Hopefully you now have a good overview of the IDE and its most important functions. Let’s have a look at the Arduino programming language next.

THE BASICS OF ARDUINO PROGRAMMING

WHAT IS THE "ARDUINO LANGUAGE"?

The Arduino language is actually 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 recognise and work with. 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 customisations. In 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. A special program, the compiler, does this compilation. The Arduino IDE ships with an open-source C++, so you don't have to worry about the details. But just 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.

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

THE STRUCTURE OF AN ARDUINO SKETCH

The simplest possible Arduino sketch is this:

(Gist for web page: https://gist.github.com/futureshocked/f124835c8d115d5825c2.js)

```
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()**". The Arduino will execute anything you put in this function 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 **loop()**, and it will continue running it in a loop, again and again, until you reset it or cut of 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. They 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 simply 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 organising 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 (gist at https://gist.github.com/futureshocked/980b11f34634cefb3359.js):

```
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), and it has no spaces or other special characters like "%", "$" and "#". You can't use a number as the first character. If in doubt, remember to only use letters, numbers, and the underscore in your function names.

The parameters passed to the function are named "**a**" and "**b**", and are both integers ("**int**"). The values that these variables contain can be accessed inside the body of the function. Parameters are optional, but if you don't need to include any you will still need to use the open/close parentheses.

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 you 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 are 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 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 in 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 you process the 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 in order 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 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	Hold 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 bytes	Can hold numbers from -32768 to 32767.
unsigned int	2 bytes	Can hold numbers from 0-65535
word	2 bytes	Same as the "unsigned int". People often use "word" for simplicity and clarity.
long	4 bytes	Can hold numbers from -2,147,483,648 to 2,147,483,647.
unsigned long	4 bytes	Can hold numbers from 0-4,294,967,295.
float	4 bytes	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 - char array	-	A way to store multiple characters as an array of chars. C++ also offers a String object that you can use instead, which offers 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 text from the Arduino displayed on your screen. More about this later. Here I just 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 nice 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.

This has benefits in terms of memory and processing speed, and is a good habit to get used to.

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 perform an… operation on 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.	int a = 5;
		a+= 2;
		
		This will result in a containing 7 (the original 5 plus a 2 from the addition operation).

++, --	Increment and decrement by 1.	int a = 5;
		a++;
		
		This will result in a becoming 6.

==, !=, <, >, <=, >=	Comparison operators. Will return a boolean (true or false) depending on the comparison result.
	
	● == → equality
	● != → un-equality
	● < → less than
	● > → greater than
	● <= → less or equal than
	● >= → greater or equal than
		int a = 5;
		int b = 6;
		boolean c = a == b;
		
		This will result in variable c contains a false boolean value.

!, &&, ||	Logical operators. The "!' operator will invert a boolean value. The "&&"
	
	! → NOT (invert) of a boolean value
	
	&& → AND of two booleans
	
	|| → OR of two booleans
		boolean a = true;
		boolean b = true;
		boolean c = false;
		
		boolean x = !a; // x → false
		
		boolean y = b && c; // y → false
		
		boolean z = b || c; // z → true

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.

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 exact 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 repeat at any given time. With this, 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**, which 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 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**". This keyword will cause the execution of the sketch to jump out of the block of code that is in 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 more 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 mentioned 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 them 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's take 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 hand

	bool finger;

	int rotation;


	//These variables change the state of the hand

	function open_finger();

	function close_finger();

	function rotate(degrees);


	//These variables report the state of the hand

	function 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 naming it "**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 is 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 is 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 it components such as these so that the programmer can think more abstractly about the thing s/he is working with instead of the implementation details.

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

```
#include <Robot_hand.h>
```

```
Robot_hand robot_hand();


void setup(){

}


void loop(){
       robot_hand.open_finger();
       robot_hand.rotate(45);
       robot_hand.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 built based on the instructions in the **Robot_hand** class, and we give it the name "**robot_hand()**". The name of the object cannot be the same as the name of the class, which 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 in order to get started with Arduino programming, this level of basic understanding of object orientation can take you a long way.

Input and outputs

Inputs and outputs 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.

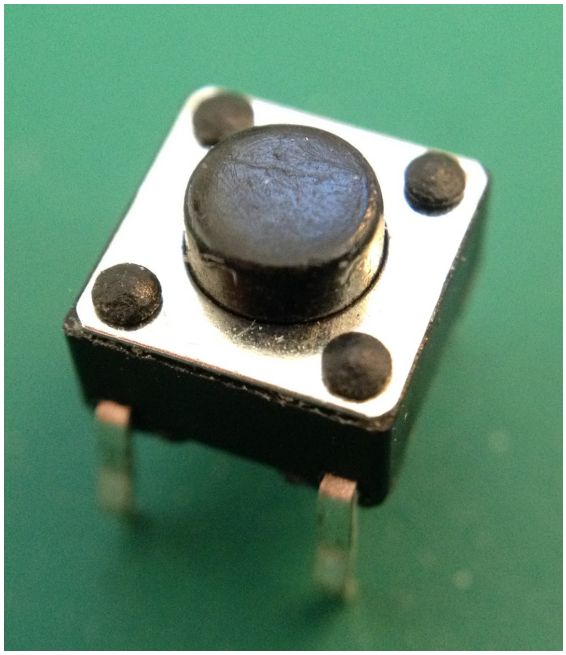
Let's have a look at them next.

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.

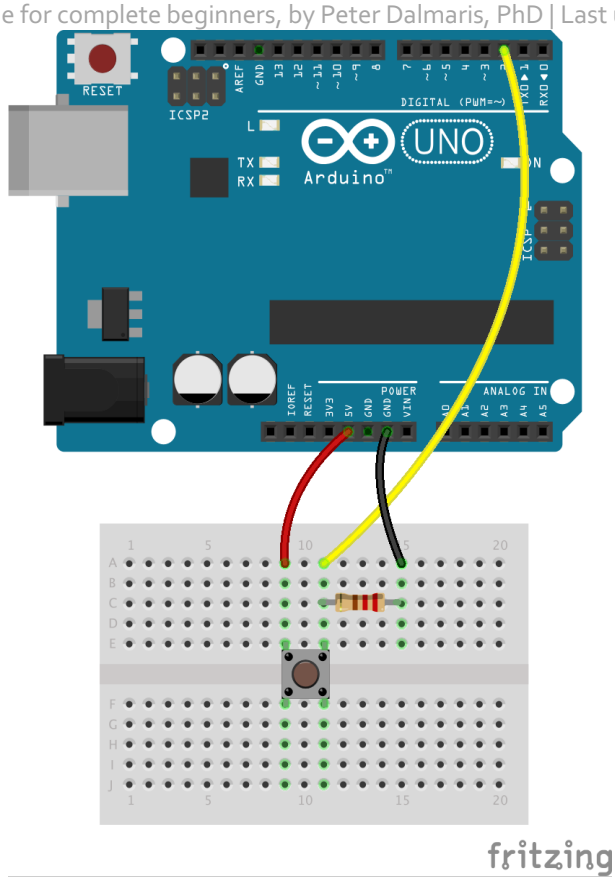
A button can be either pressed on not pressed. A switch can be on or off. A relay can be energised or not.

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:



A button is connected to digital pin 2. There is also a 10KOhm 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, and thus this level if 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)

       {
```

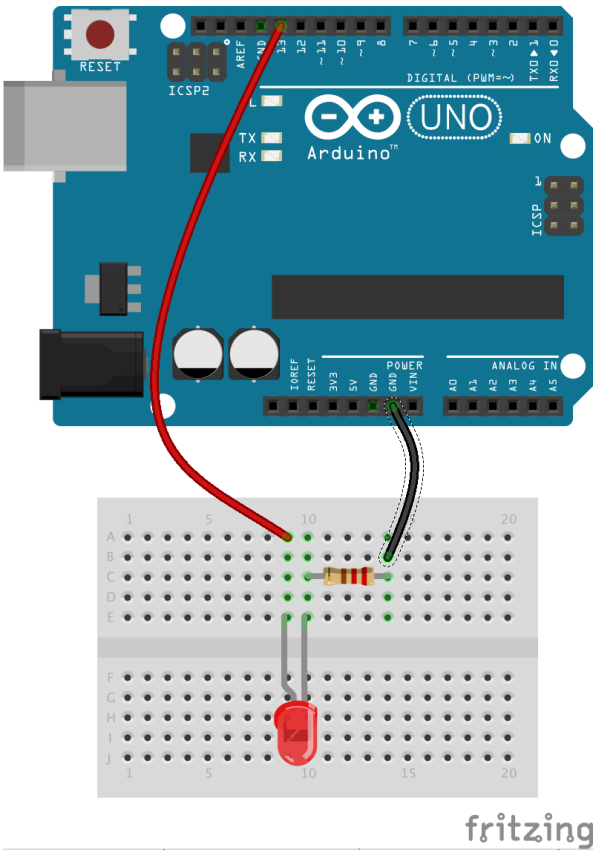
```
            //Do something when the button is pressed

      } 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 particular state by using the "**if**" conditional structure.

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



An LED is connected to digital pin 13. A 220Ohm resistor protect 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 second

  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW

  delay(1000);              // wait for a second

}
```

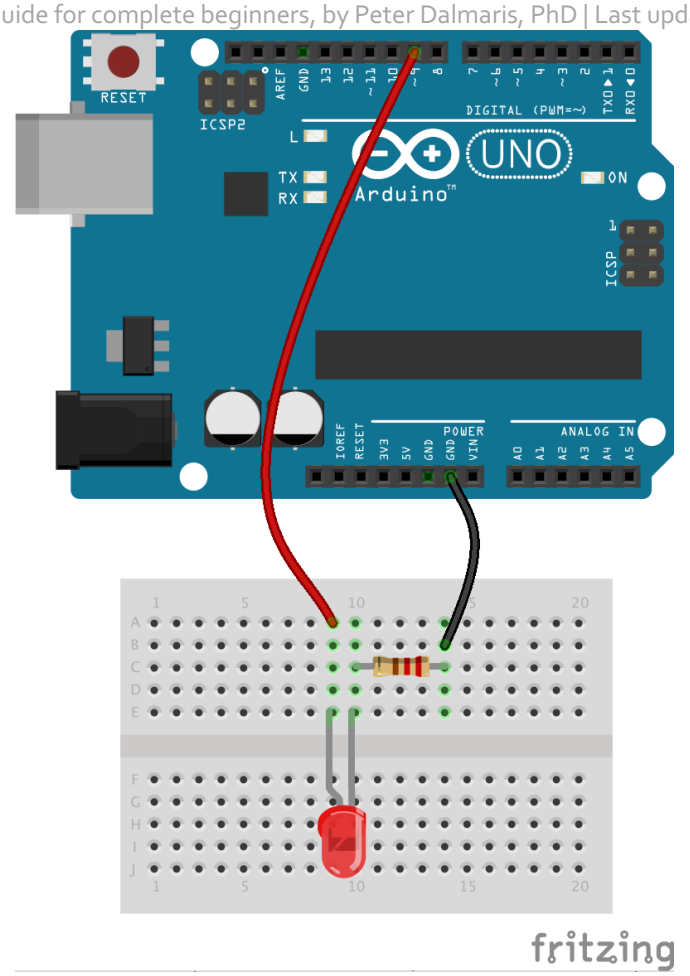
Just like with the button example, first we must tell the Arduino that we wish to use digital pin 13 as an output. We do this in the **setup()** function with **pinMode(13,OUTPUT)**.

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 are 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:



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 analogue value to a PWM pin, look at this video to see what the end result is like (Youtube, txplo.re/fadin2541):

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);
```

```
  }

}
```

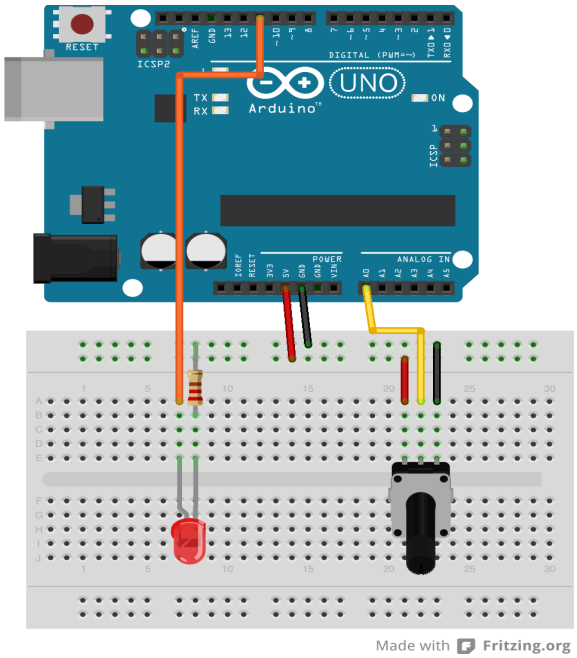
In the middle of the loop() function you will find a reference to the "analogWrite" function. This function takes two arguments: the pin number, and an 8-bit PWM value.

In the example, the variable **fadeValue** contains a number that changes between 0 and 255 in hops of 5 each time **analogWrite**. This happens because is called because it is inside a "**for**" loop. In the "for" loop definition, the parameter "fadeValue += 5" is responsible for increasing the value of the fadeValue variable by 5, each time the loop completes one iteration.

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 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.



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 become 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 get both 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 (Youtube, txplo.re/youtu4e21).

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.

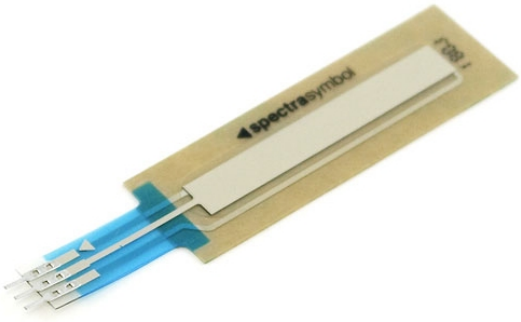
In the loop function, we get a reading from analog pin 0 (it's name is "**A0**") and store it in a local integer variable, **potValue**. The function **analogRead** returns an integer with a range from 0 to 1024. Remember from the earlier example that the **PWM** function can only deal with a value from 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.

Did you 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 like 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.

WHAT'S NEXT?

Well done for making it to the end of this introduction to the Arduino! If nothing else, it shows that you are serious about learning more about it, and, of course, for creating things with it.

There are an incredible variety of things that you can do with your Arduino.

You can build toy cars (as I show in my Arduauto course), flying drones, useless boxes, home notification systems, environment monitors (as I show in my Udemy Beginning Arduino course), remote-controlled lawn mowers, model train controllers, a satellite (you'll need to also find a rocket!), a home security system, a 3-D printer, a robot arm, a GMS phone, a hear-rate monitor, a GPS car tracker, and so many others.

Go over to Tech Explorations and find a project that is exciting and relevant to you and build it. Making is the best way of learning!

The Arduino is all about making!