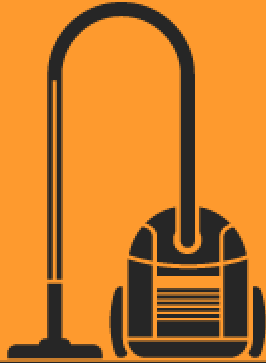


Arduino Succinctly

Marko Švaljek

Syncfusion | Technology Resource Portal

Arduino Succinctly

By

Marko Švaljek

Foreword by Daniel Jebaraj

Copyright © 2015 by Syncfusion Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

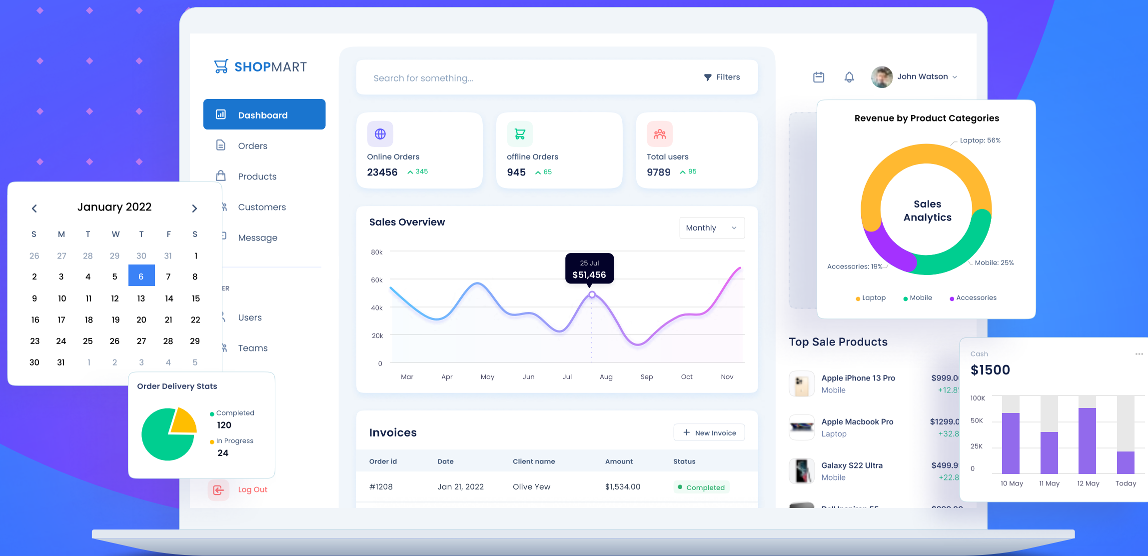
Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: Sander Rossel
Copy Editor: Suzanne Kattau
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Tres Watkins, content development manager, Syncfusion, Inc.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

1,700+ components for mobile, web, and desktop platforms

Support within 24 hours on all business days

Uncompromising quality

Hassle-free licensing

28000+ customers

20+ years in business

Trusted by the world's leading companies

Table of Contents

The Story behind the *Succinctly* Series of Books 6

About the Author 8

Chapter 1 Introduction and Getting Started 9
- Installing Arduino IDE on Windows 10
- Installing Arduino IDE on Linux 14
- Installing Arduino IDE on Mac OS X 16
- Arduino Uno Hello World 16
- Serial Monitor 20

Chapter 2 Building Circuits with LEDs 22
- Arduino Traffic Light 24
- Arduino Cylon Eye 27
- Countdown 30

Chapter 3 Working with Buttons 34
- Pushbutton 34
- Quirky Pushbutton 35
- Pushbutton with Noise Filtering Software 38
- Pushbutton with Noise-Filtering Hardware 39

Chapter 4 Using Buzzers 41
- Counting Seconds 41
- Changing Buzzer's Frequency 44
- Using the Tone Function 47
- Playing a Melody 49

Chapter 5 Measuring Environment Conditions 52
- Measuring Air Temperature 52
- Detecting Light Levels 55
- Using Temperature and Humidity Sensor 58
- Measuring Barometric Pressure 61
- Detecting Soil Moisture 65
- Measuring Environment Conditions Conclusion 69

Chapter 6 Detecting Objects 70
- Using Potentiometers 70
- Using Ultrasonic Distance Sensor 72
- Reacting On Approaching Objects 76
- Tuning the Distance Sensor on the Fly 78
- Parking Sensor 81
- Using Infrared Motion Sensor 84
- Turning the Light on Conditionally 87

Chapter 7 Networking 90
- Communication with MK Modules 90
- Using nRF24L01+ Data Transceivers 94
- Connecting to Wireless with ESP8266 Chip 100

Getting Data Online with ESP8266 Chip **105**

Chapter 8 Conclusion **110**

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click" or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Marko Švaljek works as a software developer in the machine networks department of the Kapsch Group. For the majority of his career, he didn't have much to do with electronics, and his code consisted of mostly showing something to users on a screen, or it moved data around storage systems on the backend.

One evening, out of curiosity, he ordered an Arduino board. Although he wasn't aware of it at the time, this little board changed a lot in his life. One of the first and very basic things he did with it was measure the temperature and the light level on his work desk. Having a background in software development, the first thought he had was about saving the data that came in from the sensor and analyzing it.

Sensors around the world actually generate a lot of data that is difficult to handle with the traditional technologies. So, at about the same time, he started to learn about Big Data technologies and especially about time series data (i.e., basically storing the sensor readings together with the timestamp when the measurements were made). Then he started to write about both Big Data and Electronics on his blog.

This opened a lot of doors, such as being invited to attend and talk at conferences that use those two technologies. Shortly after he published the book *Cassandra Succinctly*, he always felt that the technology that started the whole thing went a little under the radar. So he decided to write the book that will help others get into the world of hacking with the electronics and especially the Arduino board.

Chapter 1 Introduction and Getting Started

Nowadays, microcontrollers are everywhere around us. For many years, doing something with them took a lot of knowledge that had to be accumulated over a fairly long period of time. Using microcontrollers was also relatively expensive, so most of the time they were used by trained professionals in various industries.

This changed a lot back in the 2005 when the Arduino board came out. The whole idea behind Arduino is it provides hobbyists around the world with a relatively inexpensive means for building their electronics projects. A lot could be said about the Arduino, and the microcontrollers in general, but this is a succinct book so we'll keep it short and to the point. Keep in mind that there are a lot of varieties of Arduino boards out there. You can look them up on the Arduino website here.

Describing every component is outside of the scope of this book. All of the examples in this book are going to be made with the Arduino Uno board. For more details on the Arduino Uno board, see this page on the Arduino website. This board is best suited for people starting out because it is relatively inexpensive and doesn't require any soldering or special power source to get you started. Basically, all you need is an Arduino Uno board, USB cable, and a computer.

Note: You have to acquire an Arduino Uno board to follow along.

As we start using the electronic components, I will present you with a parts list at the beginning of every section where needed. The examples will be described with a sketch and a table showing how a component should be wired. In the final section, we will talk about networking. In that section, we will use two Arduino boards. Once again, one Arduino is perfectly fine and you will be able to follow along for most of the book. But at the end of the book, you will need two of them.

The examples in this book do not have any special preferences when it comes to the electronic components with which we will be using. You are free to get them from whatever source is best suited to you, financially or otherwise. The circuits that we'll be making won't use any external power source other than the Arduino itself. For someone who is just getting started with the Arduino, using strong currents might be a bit dangerous, might ruin the equipment, and might harm you. So we'll just stick with the power coming from your computer and the Arduino Uno board.

To some of you, the electronic components that we'll use might be unfamiliar because you've never heard about them, but don't worry, we'll give a simple explanation of the basic working principles with every one of those components when we mention them for the first time. If some of the concepts still aren't clear to you, please look at the Arduino website before reading further.

The purpose of having the Arduino Board is to program it to do something, so it is desirable that you have at least basic programming knowledge. It might be a bit hard for you to follow along without basic programming knowledge. Knowledge of the C programming language or any other similar language is also desirable.

We'll also try to keep the programs simple and will make many comments in the code so that you can follow along. Getting started with Arduino programming is relatively easy. There is a free Arduino integrated development environment (IDE) available for download on all of today's most popular operating systems.

We will use the IDE because it will make the upload of our programs to the Arduino board easy. Basically, there will be just a couple of clicks needed and then we'll see the magic happen as our code will actually start to control real-world objects such as lights and buzzers.

Before doing anything with the board, we have to install some software that will enable us to program it. In the next couple of sections, we'll go through the installation steps for the most popular operating systems. We'll start by using Microsoft Windows.

Installing Arduino IDE on Windows

At the moment of writing this book, Microsoft Windows 7 still seems to be the most popular Windows operating system (OS) on the market. The next most popular Windows OS is Windows XP. So we'll concentrate on how to install Arduino IDE for Windows 7. There is an installer for Windows available on the Arduino website here. Download the installer and run it. There will be a couple of steps to take in order to install it on the Windows platform:

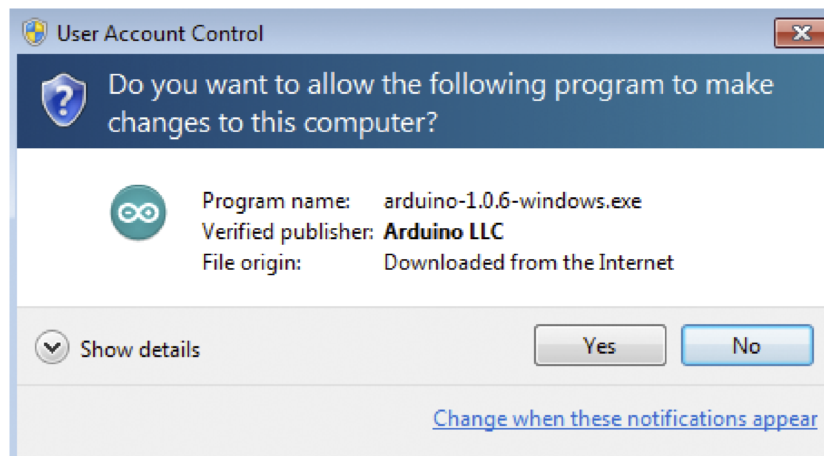


Figure 1: Allow the application to make changes to the computer by clicking Yes

Figure 2: Agree with the License Agreement by clicking "I Agree"

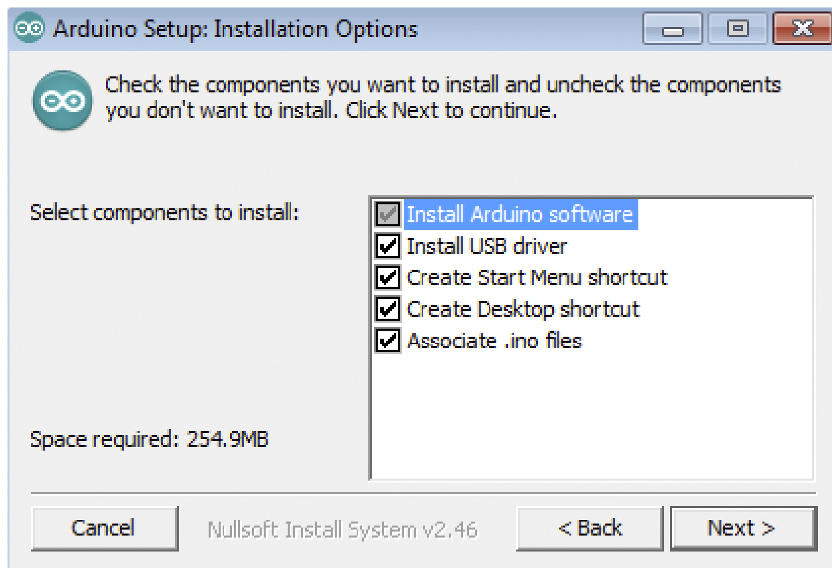


Figure 3: Select Installation Options and click Next

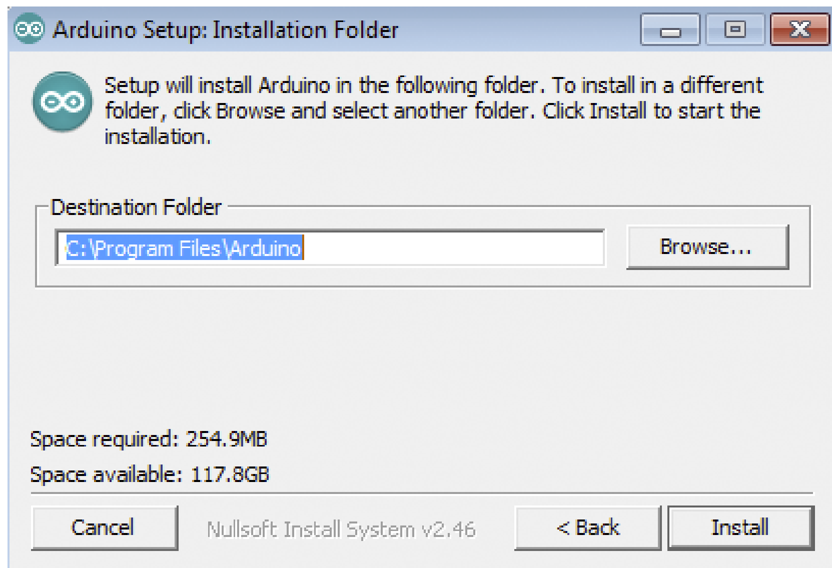


Figure 4: Select Installation Folder and click Install

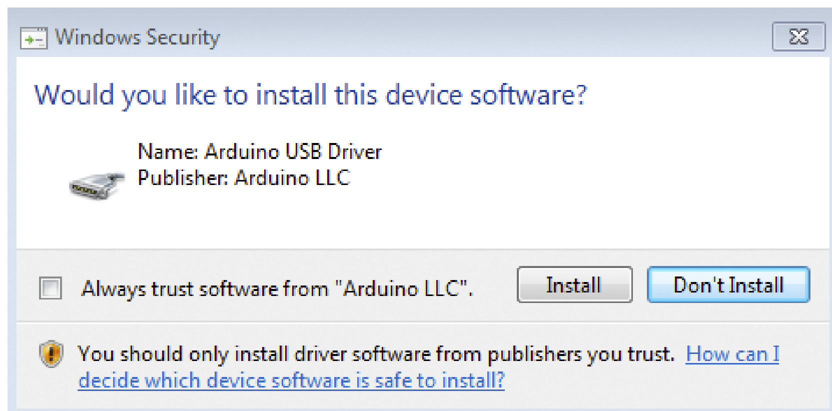


Figure 5: Allow the installation of the Arduino USB Driver by clicking Install

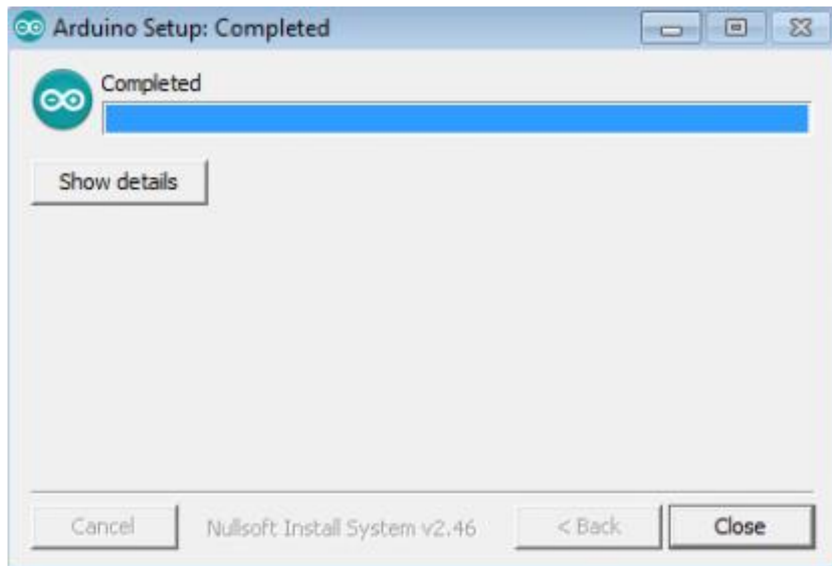


Figure 6: After the installation completes, click Close

If you followed the steps shown in previous figures, by now you should have a functional Arduino IDE on your Windows machine. Start the Arduino IDE by clicking on Arduino in the Start menu or by double-clicking the Arduino icon on the desktop. You should see something like the following:

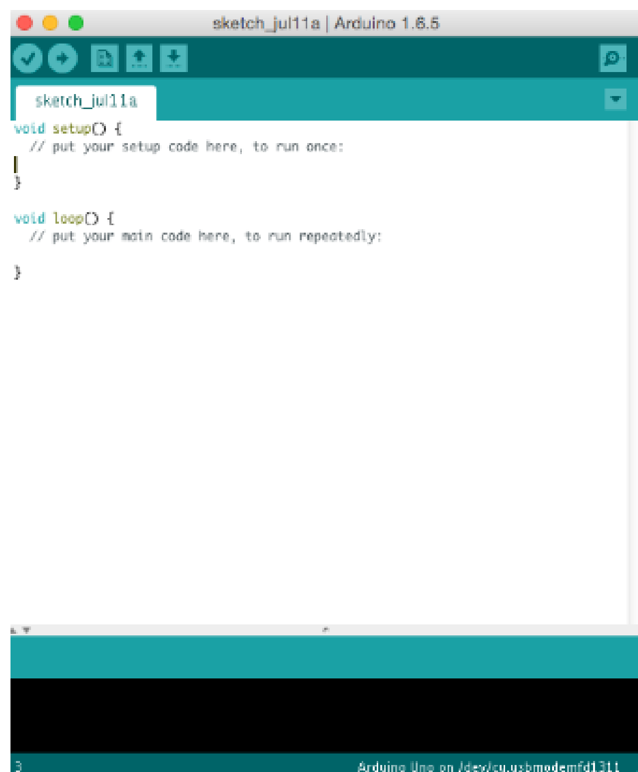


Figure 7: Arduino IDE

When you plug the Arduino in for the first time, you might wait for some time until Windows recognizes and installs the drivers for the Arduino Uno board. This is pretty common; just wait for a while, and you should be up and running with your Arduino in no time. This is as far as we'll go for now. At this stage, the most important thing is that we have an installed Arduino IDE. Later we will cover how to actually hook up the Arduino Uno board with the Arduino IDE and how to upload the code to Arduino.

Installing Arduino IDE on Linux

The installation on the Linux platform is not as straightforward as it is on the Windows platform. There is a prerequisite needed before installing and running the Arduino IDE. This prerequisite involves having an installed Java runtime environment. There are a lot of Linux versions available. Around one-third of desktop Linux machines run on Ubuntu. We'll cover how to install Arduino IDE on Linux Ubuntu in this section. The most basic Java installation for the Linux Ubuntu is to start the terminal and then run the following commands:

```
[Update the package index]
# sudo apt-get update

[Check if Java is already there, skip to Arduino installing if it's installed]
# java -version

[Install Java if it's not present on your system]
# sudo apt-get install default-jre

[install all the required packages and check if Java is installed]
# java -version
```

After you've checked if Java is present on your system (or after you installed it from scratch), it's time to download the IDE from the Arduino website here. Click the download link for the Linux version.

Be careful to check if your system is 32- or 64-bits and download the appropriate version from the website. If the versions of Linux are not matching your system, you will not be able to upload the programs to Arduino, and you will get an exception when trying to do so.

The Arduino IDE download for Linux is compressed. Open it after downloading it and extract it to a folder where you usually install your applications. Make sure that you remember the folder where you extracted the Arduino IDE, and then navigate to the folder where you extracted the Arduino IDE. Run the **`arduino`** executable file by double-clicking it. The system will ask you the following question:

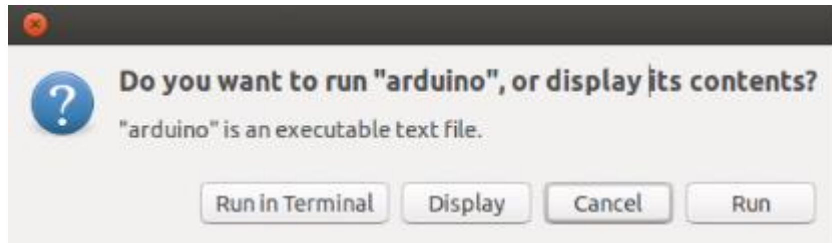


Figure 8: Click Run

If you are running the IDE from the terminal, you will not get the dialog shown in the earlier figure. There is one more step before you actually start working with the Arduino IDE. The IDE will also ask you where to store the newly created sketches. In short, sketches are source code files for programs that can be uploaded to Arduino boards:

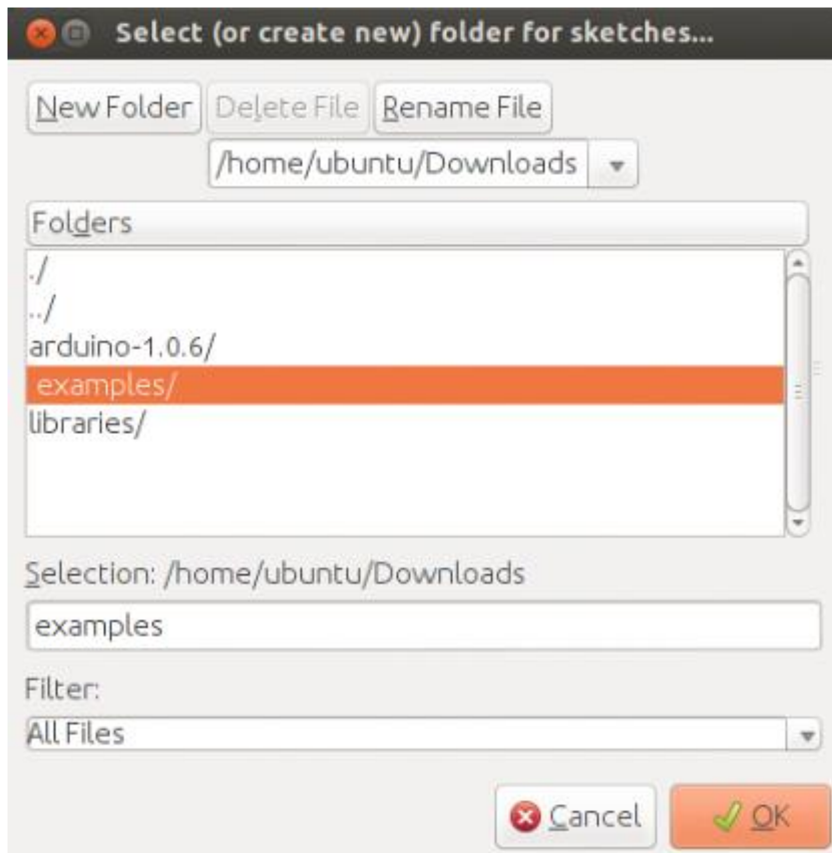


Figure 9: Specify a folder where IDE is going to save sketch files

Installing Arduino IDE on Mac OS X

On Mac OS X, Java is a prerequisite for running Arduino IDE. Java is not available on the Mac OS X by default—at least not with the most recent version. There are multiple ways you can bring Java to your Mac. Some of the previous versions of Mac had it installed by default. The easiest way to get started with Java on Mac is to go to the Apple support page available here, and download and install the provided files. As with the Linux and Windows versions, the Arduino IDE is available on the Arduino website here.

The Mac OS X Arduino IDE is compressed by default. Extract the archive to a folder and remember where you extracted it. The rest of the installation is easy; just take the extracted application and copy it to Applications folder. Then, run the Arduino the same way as you would run the other applications installed on your Mac:

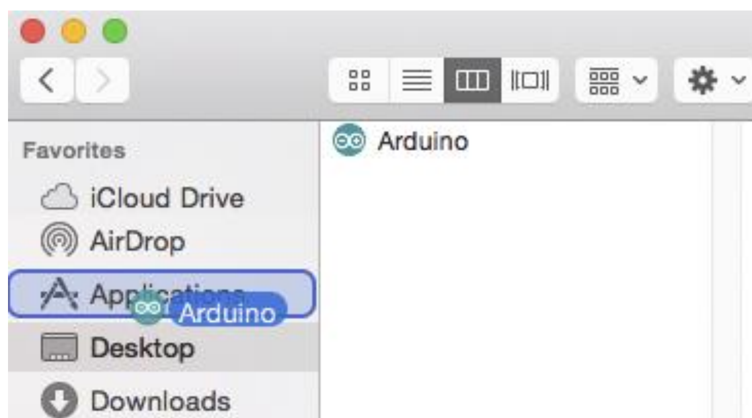


Figure 10: Drag the extracted application to the Applications folder

Once the Arduino IDE is started, it's pretty much identical no matter what platform you are using. In the next section, we will cover the basic Hello World of Arduino programming.

Arduino Uno Hello World

This is the first section in which you will need the Arduino Uno board to follow along. Besides the board, you will also need an A to Mini-B USB cable. There are two approaches that you can go by at the moment. One approach is to hook the Arduino to the USB and see what's going on when you plug it in to your computer.

A second approach is that you can get to know the parts of the board before you plug it in. Getting to know the components on the Arduino Uno board might not seem as important in the beginning, but as you progress, you will have to get to know the parts well. It might seem a bit complicated at first, but there are a couple of things of which you should be aware. Let's take a look at the Arduino:

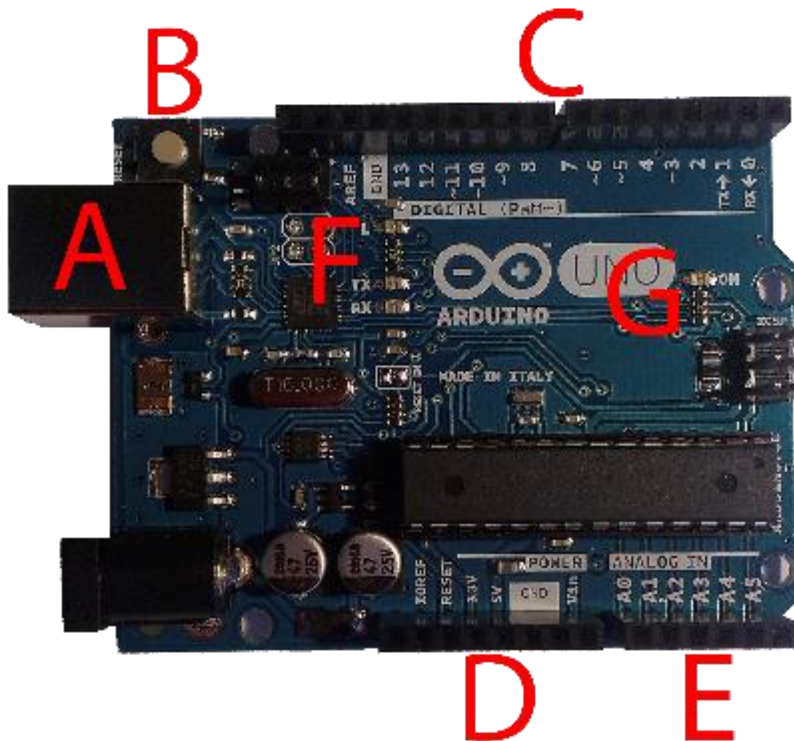


Figure 11: An Arduino Uno board

Take a moment to get acquainted with the board. Try to locate the elements located near the letters marked in the previous figure.

The following are Arduino Uno board elements that we will use:

- **A** – USB cable socket. Through this cable we power Arduino and upload programs.
- **B** – Reset button. We use this button to restart the uploaded program.
- **C** – Arduino digital pins. How these work will be explained in later chapters.
- **D** – Arduino power pins. These will be used to power electronic components.
- **E** – Arduino analog pins. How to use them will be explained in later chapters.
- **F** – TX and RX Light Emitting Diodes (LEDs) indicate Arduino communicating over USB. L is programmable.
- **G** – Power-on light. Arduino should ignite this LED as soon as it has power.

It's perfectly fine if you don't remember all of these elements in the beginning. The more you work with the board, the more you will be acquainted with it. At the moment, the only important element is the USB cable socket. Plug the USB cable to your computer and then to the USB cable socket on Arduino. Once again, you should use the USB A to Mini-B cable. If you are not sure what kind of cable that is, ask somebody working at your nearby electronics store. The cable should look exactly like (or similar to) the following:

Figure 12: USB cable type A to Mini-B

Plug the cable into the Arduino Uno and connect it with your computer. If this is the first time ever that you plugged your Arduino to the computer, you should see a small, yellow LED marked with a letter “L” going on and off at regular intervals. It’s near the area marked with a letter F on the figure that explains the basic parts of the Arduino Uno board.

This is actually an Arduino Hello World program that is running. All of the official Arduino Uno boards come with this program pre-uploaded. And this program runs as soon as the Arduino Uno board is plugged in. That way, you can quickly determine if everything is fine with your Arduino board. If, however, the LED is not going on and off, don’t worry. Perhaps you or somebody else uploaded something to it already. Anyway, we will go through the steps needed for you to get this LED blinking.

The previous section explained how to install the Arduino IDE. Please run the IDE now if you didn’t already do it. There are two steps that you need to do every time you upload a program to the Arduino Board. First, make sure the board type is set up properly in the Arduino IDE. Go to the **`Tools`** menu and then to **`Board`**, and select **`Arduino Uno`** to properly set the type.

The second step is to select the Serial Port (it’s underneath the Board menu option). Depending upon the type of OS that you have installed, the serial port might have a different name. On Windows, it will be **`COM`** followed by a suffix. On Linux and Mac, it will be something more in the direction of **`/dev/tty.xxxxxx`** or something similar. If you plugged in the Arduino Uno on Windows when the Arduino IDE is already running, you probably won’t be able to select the Serial port. Restart the Arduino IDE if this happens. Now that we have the Arduino connected, let’s do some programming. Enter the following code into the Arduino IDE:

```
// setup runs just once
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// this loop runs for as long the Arduino has power
```

```
void loop() {

  // turn the LED on with HIGH voltage level
  digitalWrite(13, HIGH);

  // wait for a second
  delay(1000);

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

  // wait for a second
  delay(1000);
}
```

We'll explain the code a bit before we continue. The onboard LED is bound to the digital pin 13. Now, you might wonder what a digital pin is. Well, there is a "digital" in its name. That means that the pin can have two states: **HIGH** and **LOW**. **HIGH** means that there is electrical current on the pin while **LOW** means that there is no electrical current on the pin. Arduino can output the current to a pin or it can read a current from a pin. In this example, we are sending an electrical current to pin 13. If the current on pin 13 is high, the onboard LED will be on. If pin 13 is in a low state, the onboard LED will be turned off.

The second part of the code is the loop function. The code inside the loop function runs for as long as there is a power source attached to the Arduino, and it will send the current to pin 13. It will wait for a second and then will stop sending the current to the pin. Then it will wait for another second and start over from the beginning. The previous code will, essentially, give the current and then take it away on pin 13 in regular one-second intervals. The unit for pausing on Arduino is a millisecond, so a one-second pause is equal to one thousand milliseconds. Now that you've typed the code, it would be a good idea to verify the code. To do that, just click Verify:

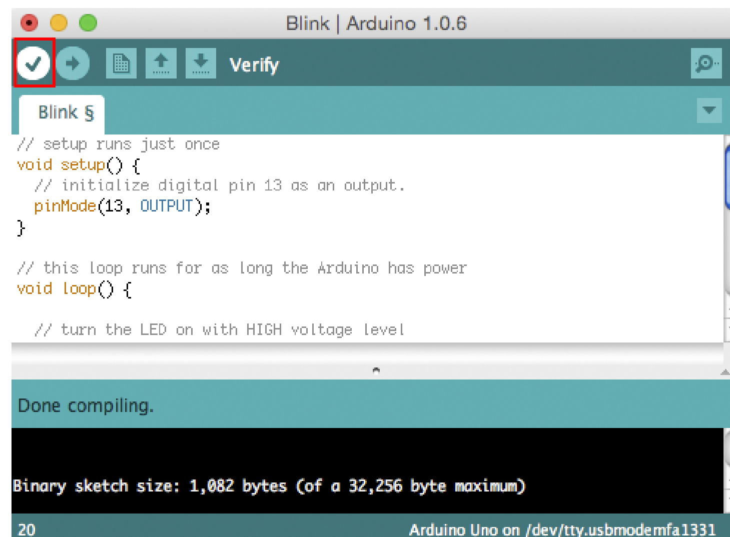


Figure 13: The Verify function in action

Now that the code is verified, it's time to upload it to the Arduino. We do this by clicking Upload. After the click, the code is compiled and uploaded:

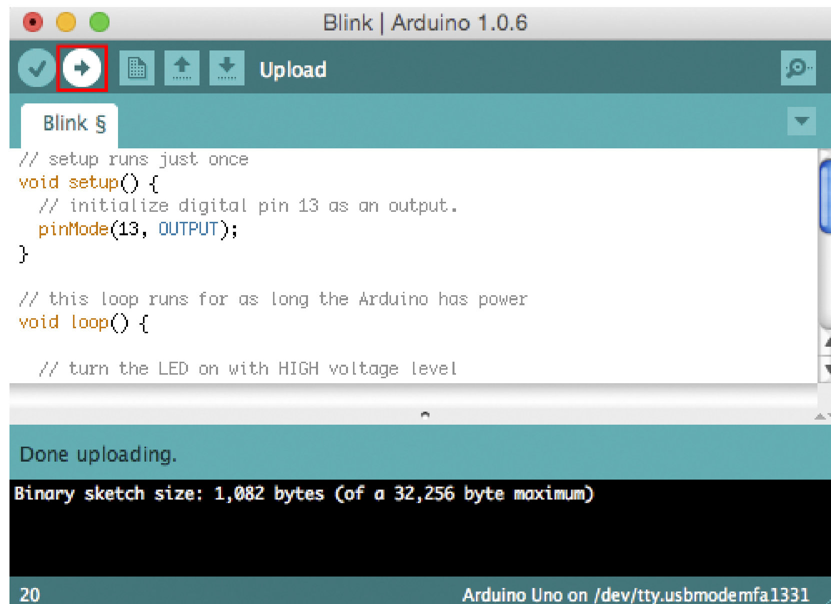


Figure 14: Upload functionality in action

If everything went fine, you should see the LED blinking. Now, it might not seem like a lot, but this is actually a fully functioning program on your Arduino board right now, and your software is actually interacting with the real world through this LED. Now, there is no useful use for this LED right now other than you learning how to control it. But the output from this pin could be used to control a lot of things. We'll talk about it through the rest of the book.

Serial Monitor

Arduino is capable of serial communication and can be used to communicate with the computer or other devices that understand serial communication. Serial communication has been around for quite some time and is often used when building various kinds of electronics projects. Perhaps it would be best to take a look at a simple Arduino example:

```
void setup() {
  // initializes serial communication with a speed of 9600 bit/s
  Serial.begin(9600);
}

// this loop runs for as long as the Arduino has power
void loop() {
  // send a message to the serial port
  Serial.println("Hello I'm your Arduino Board!");
```

```
// wait for 3 seconds and then start over
  delay(3000);
}
```

If you upload this example to your board, the board will send out a message every three seconds to the Serial port. You can read the information coming to a Serial port on your computer with various applications. You can even write your own application that communicates with the Arduino. The data interchanged in the messages is usually a comma-separated string list. This is actually a long tradition in the microcontroller world because the data is readable enough for humans, easy enough to be parsed by machines, and available without too much overhead. The downside of this approach is that the messages are not easily extensible, and every message structure change causes bigger changes in the code on both the sender and the receiver.

Arduino IDE has a nice tool that can display the data coming in from the Serial port. It's called the Serial Monitor. To access the Serial Monitor, go to the **`Tools`** menu and select **`Serial Monitor`**. If you uploaded the previous example, you should see messages appearing in the tool:

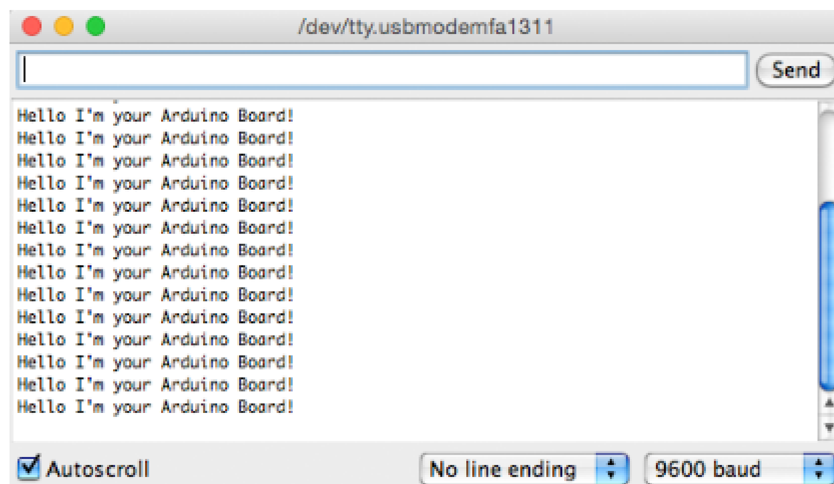


Figure 15: Arduino Serial Monitor

Aside from the communication to other devices, serial communication is also useful to debug your programs. There is an **`Autoscroll`** option with which the latest message will always be displayed. If you notice something out of the ordinary, you can disable this option and then look at the printed values.

This completes the introduction to the Arduino and the Arduino IDE. In the coming chapter, we will use electronic components—more specifically, the LEDs. Working with LEDs is relatively easy; they are relatively inexpensive components and they perform various kinds of tasks.

Chapter 2 Building Circuits with LEDs

Building circuits with LEDs is a great way to learn about electronics and the Arduino. It's relatively easy to debug everything because the LED is either on or it isn't, so if something unexpected flashes, you probably have some kind of error. In the next step, we will use just one LED. Take any LED with two legs that you have available and it should look something like this:

Figure 16: Various color LEDs

Now, most of you have heard about LEDs by now. What's specific with LEDs here is that they let the current through in one direction only, so you have to be careful when making circuits with them because they won't flash if you wire them the wrong way. Basically, you have to determine which leg is a plus and which leg is a minus. There are two ways to find this out:

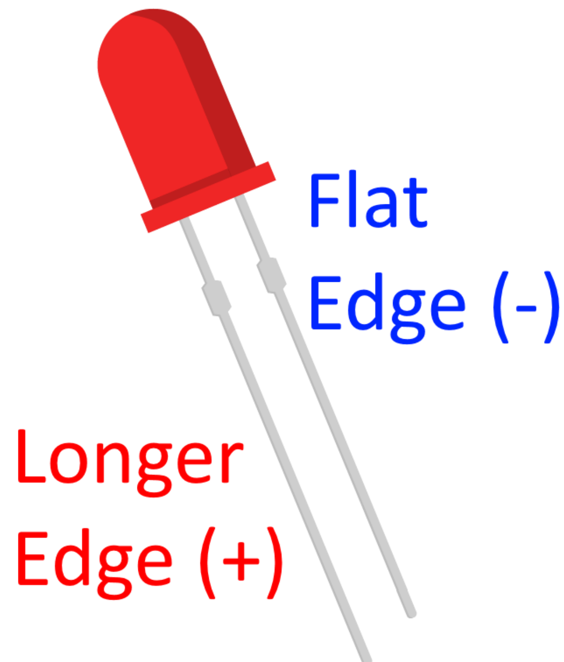


Figure 17: Determining the positive and negative lead on a LED

Most of the time, you'll be looking for a longer leg, and you'll connect it to a plus. If the legs have been cut off and you need to find out the polarity of the legs, you can find the negative leg by looking at a flat edge near the leg (on the outer casing of the LED). There is one important fact about LEDs that you have to remember. They have a bit of a self-destructive nature. They will draw as much current as possible until they burn out. To prevent this behavior, we must use an electronic element called a resistor. A resistor will limit the amount of current available to the LED and it will prevent the LED from self-destruction.

The amount of resistance a resistor is making is measured in Ohms. There is some calculation involved when determining the right resistor for the LED. You would have to look into the specifications of the LED and then fill out the sheet available here to find the right amount of resistance. With the Arduino, there are not that many voltages available. It's always either 3.3 or 5 volts. I like to stay on the safe side with LEDs so I'll be using a 100-Ohms resistor for 3.3-volt LEDs and a 220-Ohms resistor for 5-volt LEDs. You can fine-tune this to your own purposes, but generally, you'll need only two types of resistors when working with Arduino and LEDs.

To start with an actual LED, you don't need any kind of resistor at all. To begin, all you need is just an ordinary LED. If you followed along in the previous section, you've already uploaded a LED-blinking program. Now pin 13 already has a built-in resistor, so you don't need an additional one for the next example. And right next to pin 13, there is a Ground pin. The makers of the Arduino did this intentionally so that we can try out our next example. Put the longer leg called the diode into pin 13 and the shorter pin into GND. If you are not sure about the location of the digital pin 13, flip the Arduino sideways and insert the legs into holes:

Figure 18: An Arduino Uno board viewed sideways

If you properly inserted the LED, you should see it blinking in the same rhythm as the onboard LED. Your Arduino is now controlling an electronic component. If it’s a bit dark in the room, turn the light off and enjoy the scene. But this is only the beginning. We’ll build on top of that in the next section.

Arduino Traffic Light

Parts list for this section:

- Arduino Uno
- USB Cable
- 3x 100-Ohms resistor
- Breadboard
- 1x 5mm Red LED
- 1x 5mm Yellow LED
- 1x 5mm Green LED
- 4x Breadboard jumper wire

The wiring for the example looks like this:

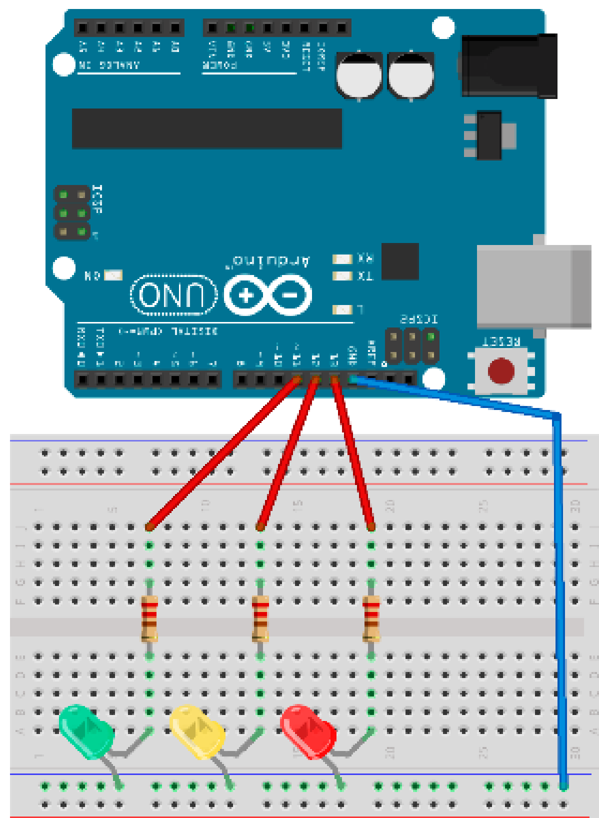


Figure 19: Arduino Uno traffic light wiring

We will connect the red LED to pin 13, the yellow LED to pin 12, and the green LED to pin 11. We will put around the same amount of time for the red and the green light. The transition from red to green will be a bit faster. The transition from green to red usually takes a bit longer. Let’s look at the code:

```
void setup() {
  // initialize red light pin
  pinMode(13, OUTPUT);
  // initialize yellow light pin
  pinMode(12, OUTPUT);
  // initialize green light pin
  pinMode(11, OUTPUT);
}

void loop() {
  // turn the red light on
  digitalWrite(13, HIGH);
  digitalWrite(12, LOW);
  digitalWrite(11, LOW);
  // wait
  delay(5000);

  // turn the yellow light on together with red
  // at least that's the way they work in some parts of Europe
  digitalWrite(13, HIGH);
  digitalWrite(12, HIGH);
  digitalWrite(11, LOW);
  // wait
  delay(1000);

  // turn the green light on
  digitalWrite(13, LOW);
  digitalWrite(12, LOW);
  digitalWrite(11, HIGH);
  // wait
  delay(5000);

  // turn the yellow light on
  digitalWrite(13, LOW);
  digitalWrite(12, HIGH);
  digitalWrite(11, LOW);
  // wait
  delay(1000);

  //start over
}
```

While you upload the code to the Arduino board, the red LED will be blinking. This is intentional so that you know that pin 13 goes randomly up and down while the program is uploading. This way, you don't attach some moving or sensitive component to it in the future. For our traffic light, this simply brings more fun to the uploading. When the program starts to run, the first light will be red. Then the yellow light will be shown. In Europe, the red light goes on together with the yellow one before the traffic light shows the green light. We will emulate this behavior just so that you can see that two LEDs can be on at the same time. Arduino will continue to switch from red to green light and back again, for as long as it has power.

There is one very important component that's new in this example, and that's the breadboard. The breadboard is a useful component because a person does not have to know much about soldering to get started with the electronics. It certainly helped me a lot when I was starting out. One other advantage of using a prototyping board is that it doesn't take long to build basic circuits, and you can rearrange components quickly if something is wrong. Under the holes of a prototyping board are wires, such as this:

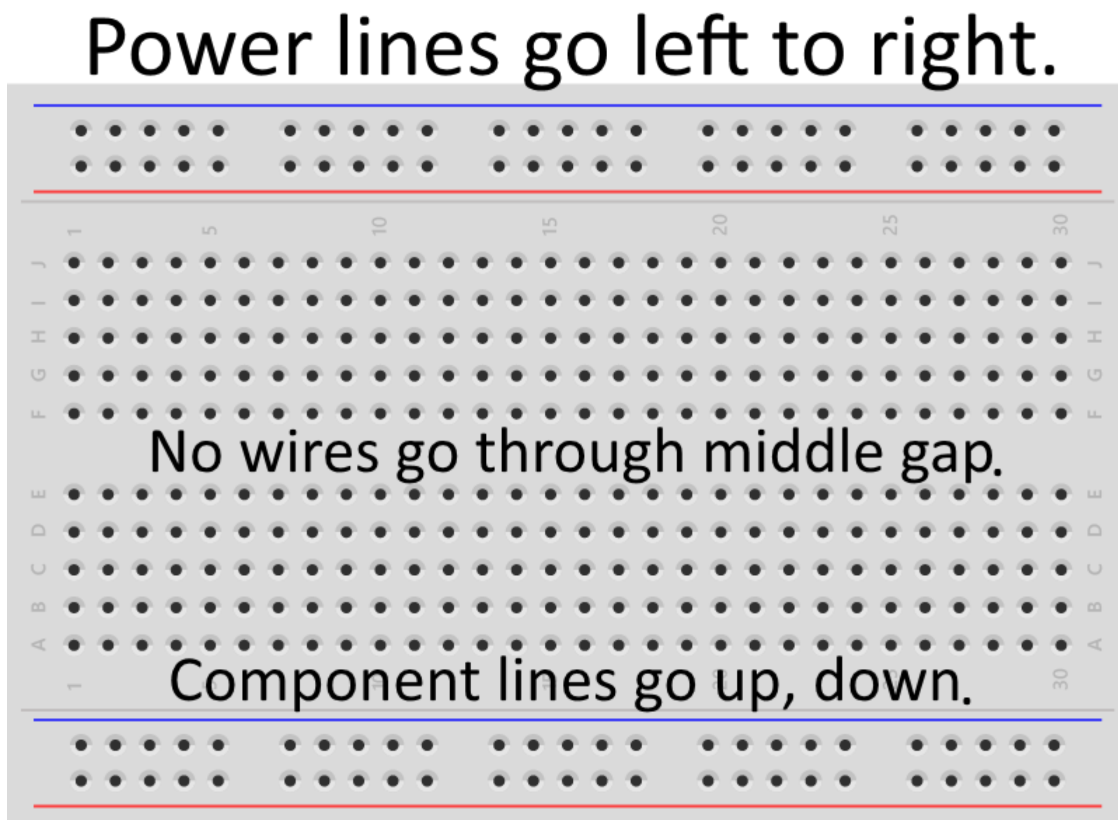


Figure 20: Prototyping board wiring

The downside of a prototyping board is that it can't accommodate a lot of components and that, with soldering, our circuits will be a lot smaller. But, for someone just going into building electronic circuits, it's a perfect starting point. It's hard to summarize the do's and don'ts of using a prototyping breadboard. You will get a feel for how to use the prototyping board as you progress, by looking at the examples in future sections.

The assumption is that most people reading this book will come from a software development background. There is one important thing we need to point out when it comes to the breadboard (and to electronic circuits in general). Some bugs are harder to find because, aside from the software component, there is a possibility of wrong wiring. Some of the wrong wiring might come from misunderstanding how the lead wires are connected in the prototyping board.

One of my biggest misunderstandings was when I bought a breadboard with multiple power lines but they didn't go all the way from the left to the right of the board. They have a break in the middle of the power lines, and that is how the longer breadboards work. If you are unaware of this fact, you might expect that the components will be under power. But they wouldn't work because power rails stop in the middle and you have to connect them with additional wires to ensure that they get the current. You can, potentially, lose a lot of time until you find what's actually going on. There are versions of a breadboard where there is no break in the middle but, just so that you are aware of the gap between the power lines on bigger breadboards. I'll depict the gap with a red line on the following figure:

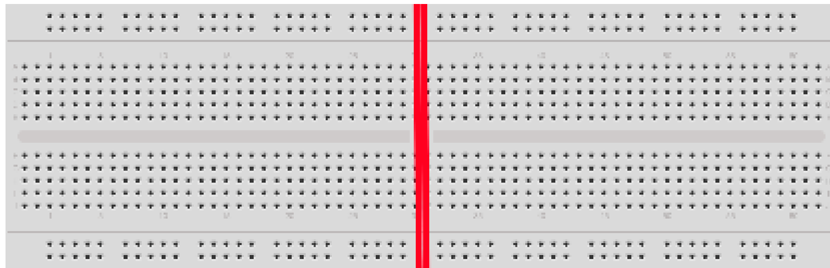


Figure 21: Un-bridged gap between the power lines in the middle of a bigger breadboard

Arduino Cylon Eye

Parts list for this section:

- Arduino Uno
- USB cable
- 10x 100-Ohms resistor
- Breadboard
- 10x 5mm red LED
- 1x Breadboard jumper wire; up to 11 wires if your resistors have short leads

I won't talk much about the Cylon Eye effect; look it up if you are not familiar with it. From an electronics angle, it's a red light going constantly from left to right. I will try to save some wires with this example, so I'll show you how to make a version with just one wire for the whole wiring. But you can also use one wire per LED as in the previous example; it's completely up to you. Here's an example with just one wire:

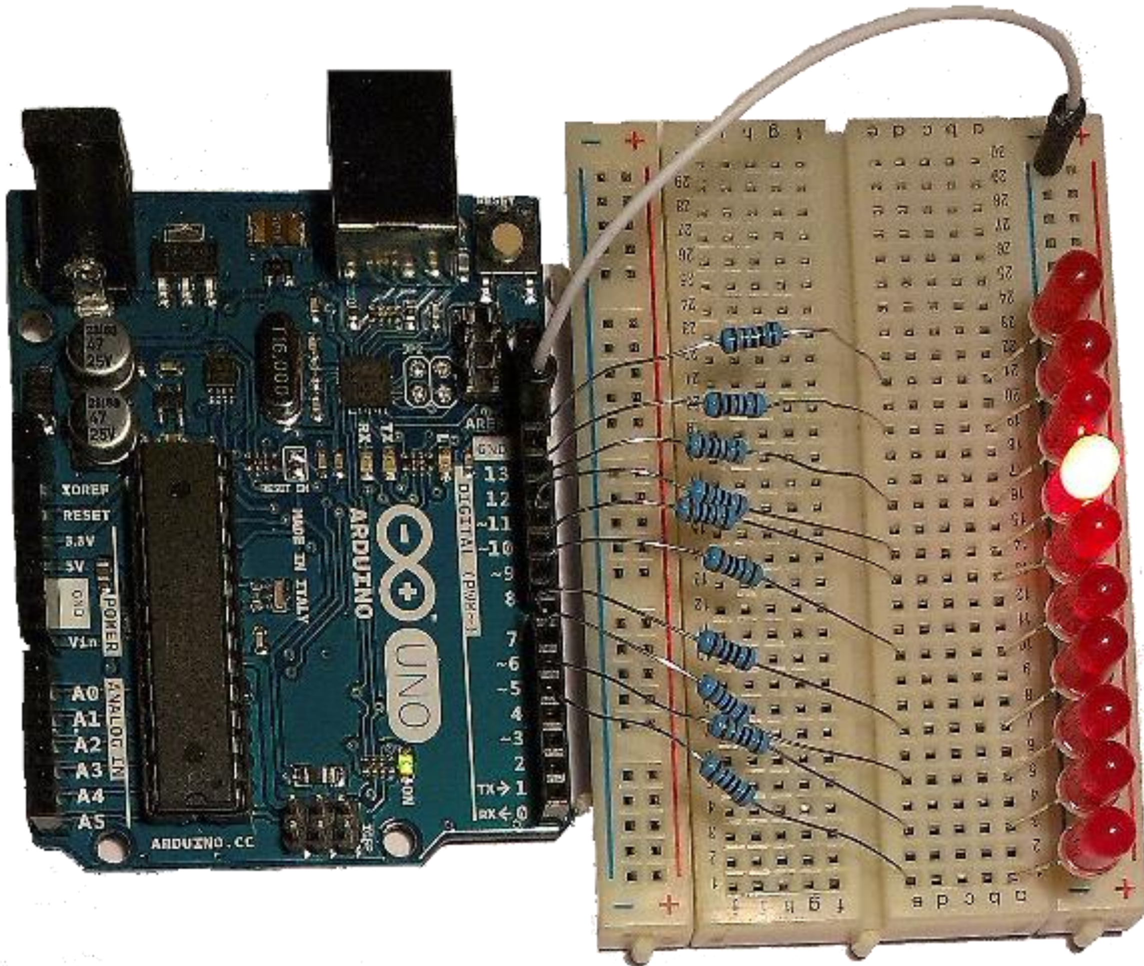


Figure 22: Cylon Eye version with a single jumper wire

The first LED is plugged into an Arduino digital pin 4. The last one is digital pin 13. Do not forget to connect every pin to a resistor before you hook it up to the LED. Also, don't forget to double-check the polarity of the LEDs because the example will not work at all if you connect the negative LED leg to the pin output. You might be wondering what's wrong when, in fact, all you did was turn the LED the wrong way. This happens a lot, especially if somebody is just starting out. Because of the viewing angle, it might be a bit hard for you to read the pin connections from the previous figure, and to know where you have to insert the leads from the components. So here's a complete schematic to help you. Just as a precaution, remember, there are 10 LEDs in a row altogether:

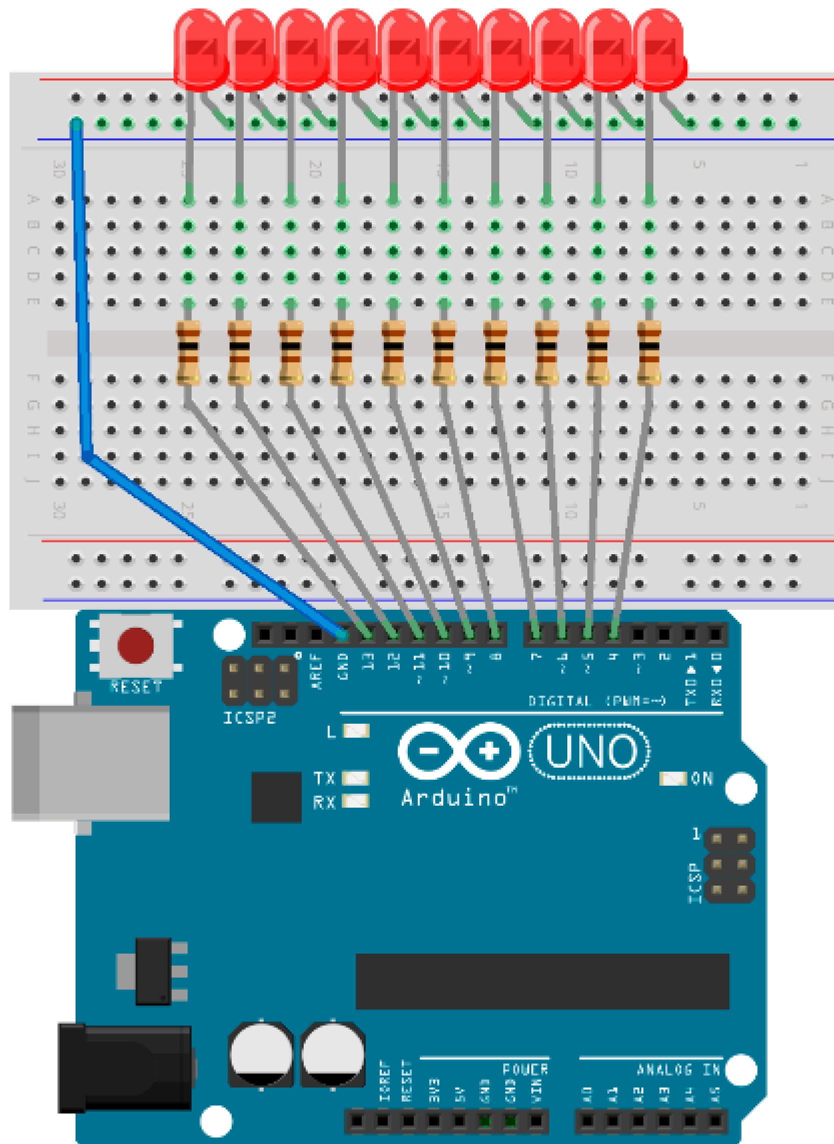


Figure 23: Cylon Eye wiring schematics

The number of LEDs that we control in this example is significantly larger than in the previous section. We'll have to use looping construct when controlling all of the LEDs. Here is the code:

```
// define the starting pin as a constant
const int startPin = 4;
// total number of LEDs
const int numLEDs = 10;

// current LED will go from 0-9
int currentLed = 0;
// direction will be 1 and -1
int direction = 1;
```

```
void setup() {
  // initialize the pins with a for loop
  for (int i = startPin; i < startPin + numLEDs; i++) {
      // set pin mode to output
      pinMode(i, OUTPUT);
  }
}

void loop() {
  // turn the led on
  digitalWrite(startPin + currentLed, HIGH);
  // wait for 100 ms
  delay(100);
  // turn the led off
  digitalWrite(startPin + currentLed, LOW);

  // move to the next LED
  currentLed += direction;

  // go back up if we've reached the bottom
  if (currentLed < 1) {
    direction = 1;
  }
  // go back down if we've reached the top
  if (currentLed >= numLEDs - 1) {
    direction = -1;
  }
}
```

I think this example looks best if viewed in the dark. So take some time to enjoy it. This example can serve as a good starting point for a great range of special LED light effects. Imagine you could change the color of the LEDs and put them in some random order when turning on and off. During the holiday season, you could give this as a gift. I don’t know why, but the light effects somehow always make people smile. If you built this example, try to play a bit with it and make various effects. In the next section, I will take it a step further.

Countdown

This example will be pretty much as complex as it gets on a small solder-less breadboard. We will use significantly more wires than in the previous sections. The number of LEDs is going to increase as well.

Parts list for this section:

- Arduino Uno
- USB cable
- 13x 100-Ohms resistor
- Breadboard

- 13x 5mm LED
- 18x Breadboard jumper wire

In this section, we are going to build a digit display out of 13 LEDs. Here's the initial idea:

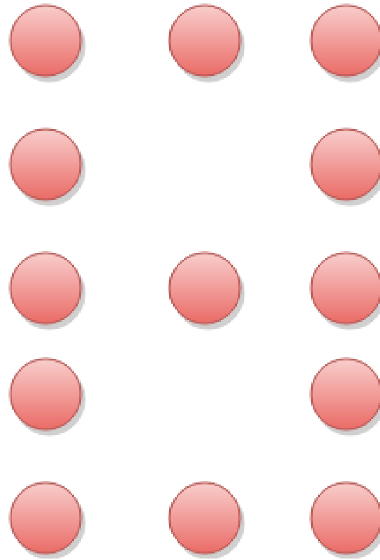


Figure 24: 13 LEDs in a single-digit display

And here it is in action where all of the 10 digits are displayed in a countdown:

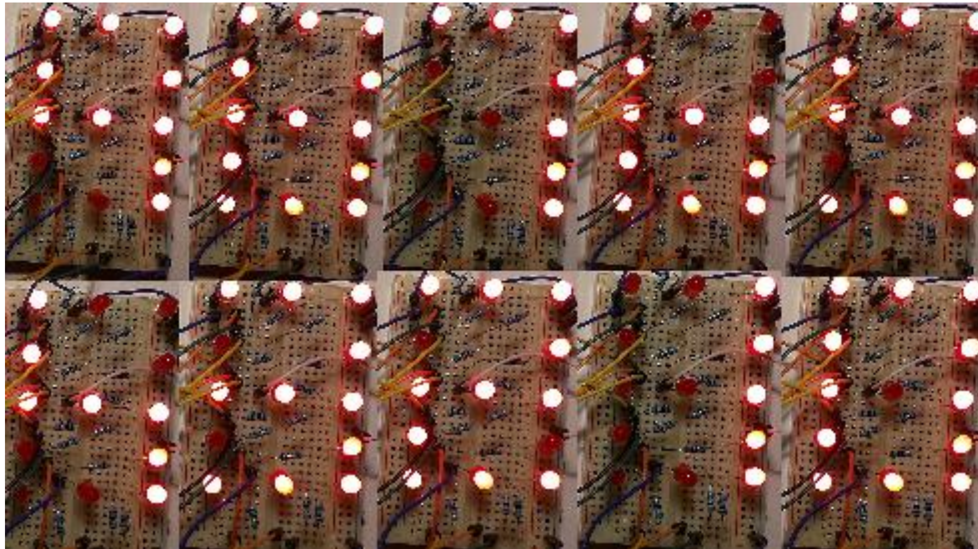


Figure 25: 13 LEDs countdown in action

The wiring of this example is a bit complex, especially if you are just beginning with the breadboard. Try to remain focused while building the example. You can rewire the example however you see best. Here is my complete wiring schema:

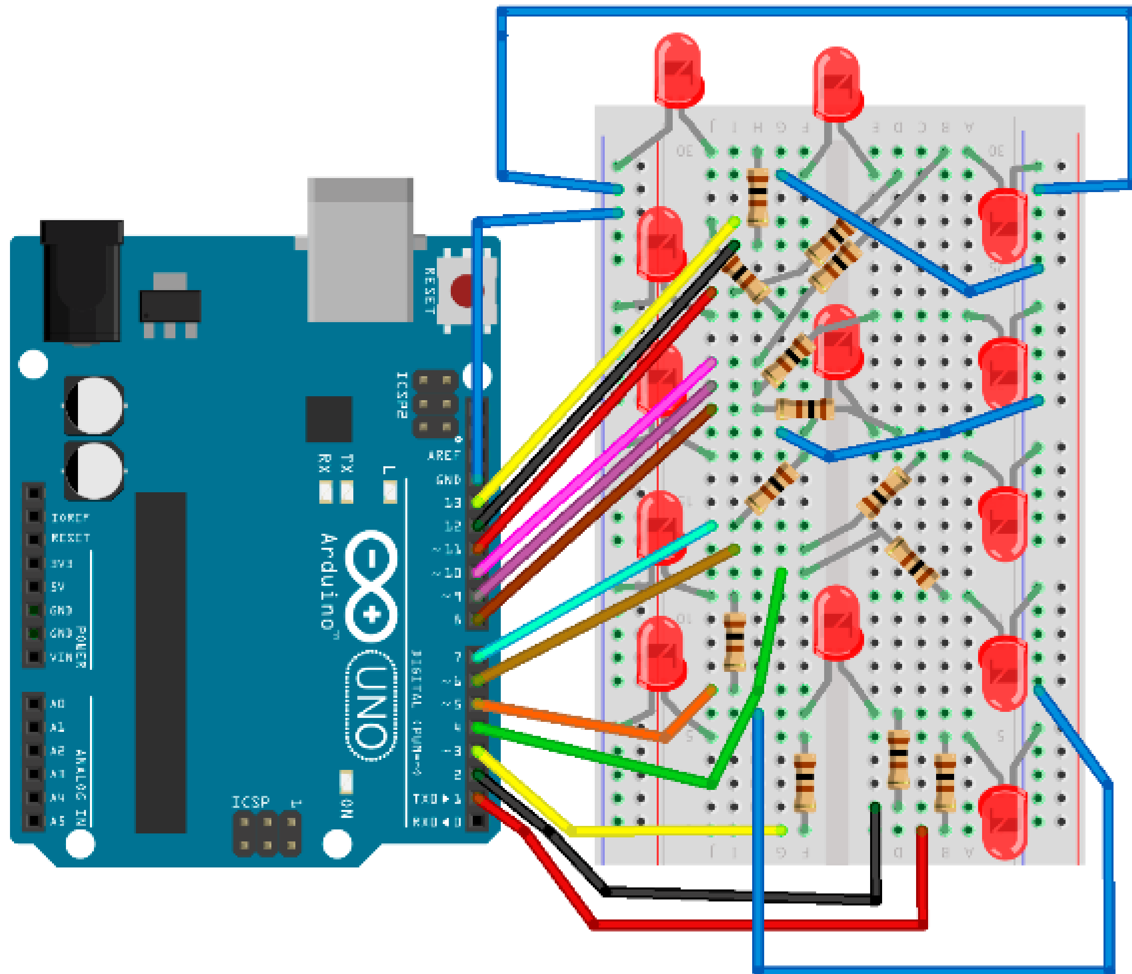


Figure 26: Countdown example wiring schema

This example is more for practicing. In practice, you would use a seven-segment display. But this is a section in which we are studying LED behavior, so we’ve built this example with LEDs and not seven segment displays. However, there is one more thing that you need in order to complete this example, namely, the code:

```
// define the starting pin as a constant
const int startPin = 1;

// total number of LEDs
const int numLEDs = 13;

// we'll start the countdown at 9
int count = 9;
```

```
// here is each digit mapped to the state of the pins
int numberMatrix[][13] = {
  {1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1}, // 0
  {1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0}, // 1
  {1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0}, // 2
  {1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0}, // 3
  {1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1}, // 4
  {1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1}, // 5
  {1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1}, // 6
  {1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0}, // 7
  {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, // 8
  {1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1}, // 9
};

void setup() {
  // initialize the pins with a for loop
  for (int i = startPin; i < startPin + numLEDs; i++) {
      // set pin mode to output
      pinMode(i, OUTPUT);
  }
}

void loop() {

  // go through the pins for the current count, turn them on an off
  for (int i = 0; i < numLEDs; i++) {
    digitalWrite(startPin + i, numberMatrix[count][i]);
  }

  // wait for a second
  delay(1000);

  // count down
  count--;

  if (count < 0) {
    // start over
    count = 9;

    // turn all the LEDs off
    for (int i = startPin; i < startPin + numLEDs; i++) {
        digitalWrite(i, LOW);
    }

    // wait for a second and start over
    delay(1000);
  }
}
```

Chapter 3 Working with Buttons

Every now and then, you will want to interact with the user. The most common element for interacting with the user is a button. The most common type of button used in electronics nowadays is the push button. This button is not closing the circuit constantly but only for as long as it is pressed. The most basic example of using the pushbutton is turning the LED on when the button is pressed. I will describe this in the next section.

Pushbutton

This example is the Hello World when working with buttons. Basically what it does is it turns the LED on when a button is pressed and then turns it off once the button is released. Here is the parts list and the wiring:

Parts list for this section:

- Arduino Uno
- Pushbutton
- USB cable
- 1x 100-Ohms resistor for the LED (pin 13)
- 1x 10k Ohm resistor for the Pushbutton
- Breadboard
- 1x 5mm LED
- 5x Breadboard jumper wire

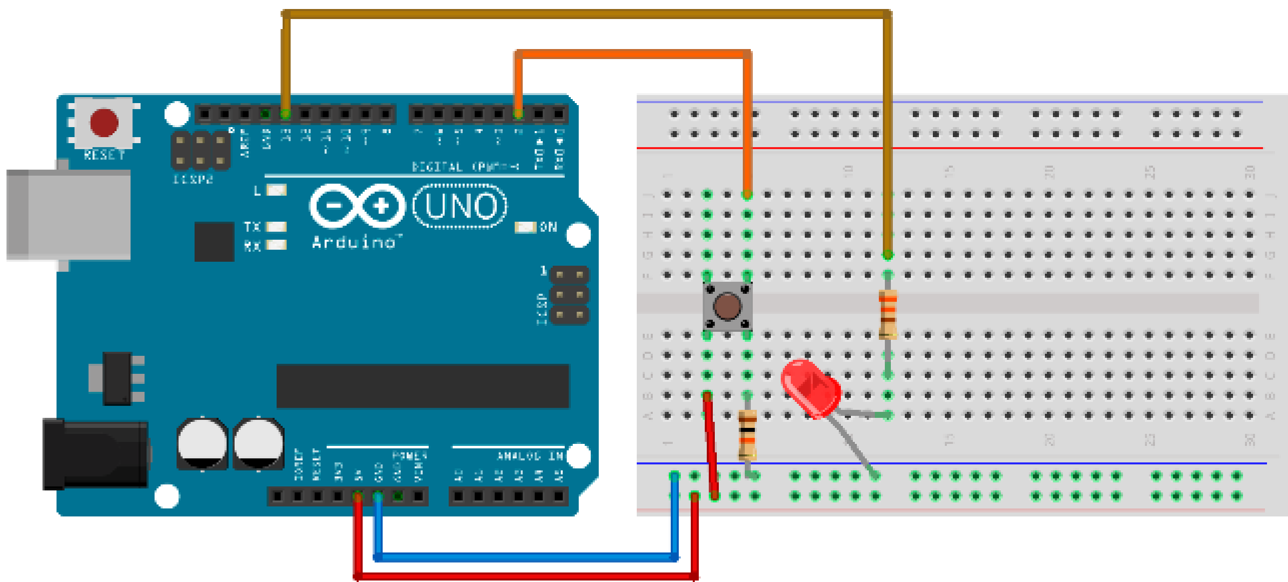


Figure 27: Pushbutton wiring schema

The code for the example is pretty straightforward. We are constantly reading the state from pin 2 and, if it’s in a high state, we turn the LED on. Note that we connected the LED to pin 13 so that the onboard LED will shine too. Here’s the code:

```
// variable holding button state
int buttonState = 0;

// LED pin
int ledPin = 13;

// Button pin
int buttonPin = 2;

void setup() {
  // initialize pins
  pinMode(ledPin, OUTPUT);
  pinMode(buttonPin, INPUT);
}

void loop(){
  // read the state of the button
  buttonState = digitalRead(buttonPin);

  // pressed button is in state high
  if (buttonState == HIGH) {
    // LED on
    digitalWrite(ledPin, HIGH);
  }
  else {
    // LED off
    digitalWrite(ledPin, LOW);
  }
}
```

In this example, we will read the pin input for the first time. We are doing it with the **digitalRead** function. The loop that checks the reading is a lot faster than the human eye can notice, so we are not noticing an interesting effect with the button. We see the LED immediately on and can’t perceive any oscillations while the button is initially pressed. Also, we don’t see that the LED is blinking for a while after we release the button. Let’s look at the next example and go deeper into this interesting effect.

Quirky Pushbutton

The wiring and the parts for this section are the same as in the previous one (so please have a look at those schematics). The difference is in the programming and in the use. For instance, turning the LED on and off is not the most useful case when working with buttons. When microcontrollers are involved, it’s usually some sort of a state change. So we will change the LED state when we press the button. To do this, we will use the following code:

```
// variable holding button state
int buttonState;

// variable holding previous button state
int previousButtonState = LOW;

// variable holding the LED state
int ledState = HIGH;

// LED pin
int ledPin = 13;
// Button pin
int buttonPin = 2;

void setup() {
  // initialize pins
  pinMode(ledPin, OUTPUT);
  pinMode(buttonPin, INPUT);
}

void loop(){
  // read the state of the button
  int reading = digitalRead(buttonPin);

  // if the button is pressed, change the state
  if (reading != buttonState) {
    buttonState = reading;

      // toggle the LED state only when the button is pushed
      if (buttonState == HIGH) {
        ledState = !ledState;
      }
  }

  // turn the LED on or off depending on the state
  digitalWrite(ledPin, ledState);

  // save current reading so that we can compare in the next loop
  previousButtonState = reading;
}
```

Now, try to play with this button for a while. If you do so, you will notice that it's not behaving as you would expect; it sometimes misses the button press, behaving a bit quirky. You will notice the light remains on even when it shouldn't and vice versa. Although we have a **digitalRead** function, there is an interesting analog effect at work here.

The switch is made of metal parts and the button press brings them together. There is a movement associated with this action, and the metal surfaces inside the switch bounce away and then come back together again a couple of times (within a very short period of time). This is a known effect and it does not represent a problem when, for instance, turning a light bulb on and off. It didn't represent a problem in our first example with the button where the LED was on for as long as the button is pressed. But, when it comes down to logical circuits where we can read the changes really fast during a short period of time, we will not be sure if the button was pressed or not.

Let's look at the following picture. Essentially, every change in the spike will cause the LED state to change. So, using this button without some kind of noise filtering is pretty much playing a lottery. And you won't be sure if the LED is going to turn on or off when you release or press the button. Here's a figure showing what's going on with the electrical current during a short button press. Remember, every change in the spike is a change in the state of the LED:

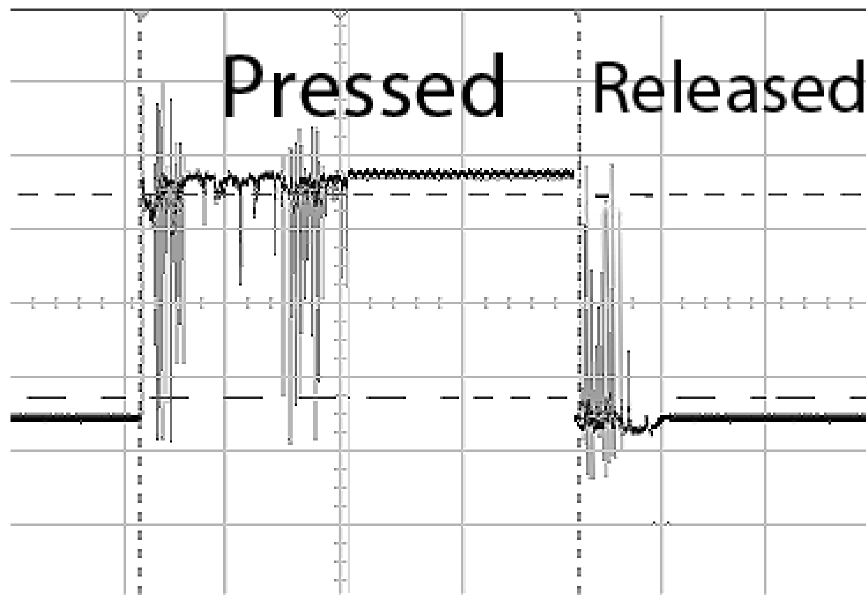


Figure 28: Electricity reading during a short button press, followed by a release

The human eye won't perceive every single change but we will notice faulty state changes at the end of a button press. In the next section, we will implement a filtering technique to avoid the quirky behavior. The circuit and the parts will remain the same as in this section.

Pushbutton with Noise Filtering Software

The parts list and the wiring are the same for this as they were in the previous example. The difference is in the software that filters the pushbutton noise:

```
// variable holding the button state
int buttonState;

// variable holding previous button state
int previousButtonState = LOW;

// variable holding the LED state
int ledState = HIGH;

int buttonPin = 2;
const int ledPin = 13;

// millis always stored in long
// becomes too big for int after 32,767 millis or around 32 seconds

// timestamp of a previous bounce
long previousDebounceTime = 0;

// debounce time in millis, if the LED is still quirky increase this value
long debounceDelay = 50;

void setup() {
  pinMode(buttonPin, INPUT);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // for now it's the same as in previous example
  int reading = digitalRead(buttonPin);

  // if the previous reading is different than the previous button state
  if (reading != previousButtonState) {
    // we'lll reset the debounce timer
    previousDebounceTime = millis();
  }

  // if the reading is the same over a period of debounceDelay
  if ((millis() - previousDebounceTime) > debounceDelay) {
    // check if the button state is changed
    if (reading != buttonState) {
      buttonState = reading;

      // toggle the LED state only when the button is pushed
      if (buttonState == HIGH) {
        ledState = !ledState;
      }
    }
```

```
  }

  // set the LED:
  digitalWrite(ledPin, ledState);

  // save current reading so that we can compare in the next loop
  previousButtonState = reading;
}
```

Now, to some people, the code listing might seem as a bit too complicated, and that's perfectly understandable. There is a neat hardware trick that we can use to filter out the noise. We can use an electronic component called a capacitor.

Pushbutton with Noise-Filtering Hardware

We will now use almost the same components and wiring as we did with the quirky and basic pushbutton example. To filter out the noise, we will add a capacitor to the circuit.

Parts list for this section:

- Arduino Uno
- Pushbutton
- USB cable
- 1x 100-Ohms resistor for the LED
- 1x 10k-Ohm resistor for the pushbutton
- Breadboard
- 1x 5mm LED
- 5x Breadboard jumper wire
- 10uF Capacitor

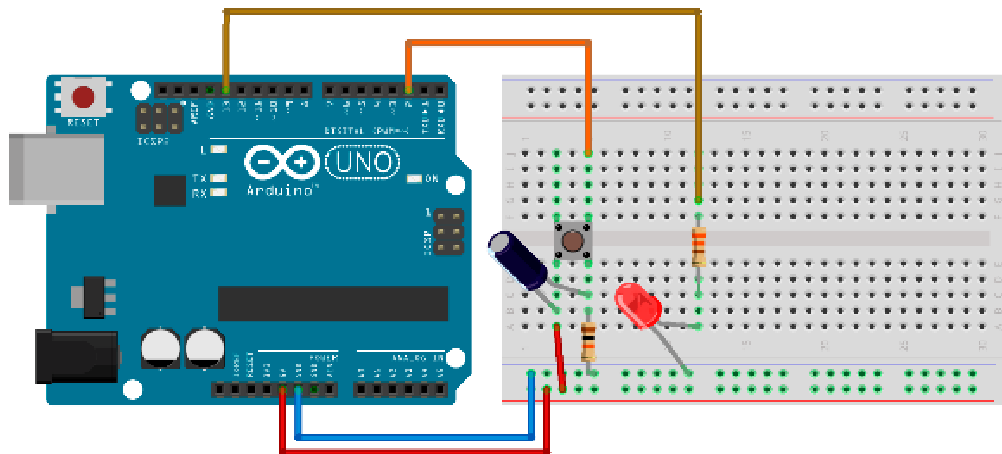


Figure 29: Pushbutton with capacitor filtering out the noise

The programming is the same as with the quirky pushbutton. The noise-filtering assignment is left to the capacitor. The capacitor has an interesting feature where it lets the current through for as long as it doesn't get filled with electricity. When it gets filled, the current doesn't flow through it anymore. But, if some fluctuation starts to happen, it immediately starts to release the accumulated current out of it (and basically smoothens the signals out). Actually filtering the electrical noise is one of the main usages of capacitors. There are various types and sizes of capacitors. Here are some of the basic capacitor forms:

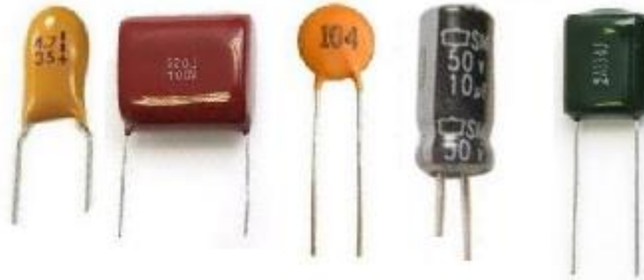


Figure 30: Basic capacitor forms

There are types of capacitors for which it's not important how they are wired. For some, you have to be careful when connecting the positive and the negative pin. On capacitors, the positive pin is marked with a plus sign and a negative pin is marked with a "0" sign. So there are no tricks for finding out the pin polarity like there are with the LEDs. There are also some types of capacitors that have a longer leg, and that leg is the plus pin; it's the same rule as with the LEDs. But a positive or a negative pin on a capacitor needs to have a clear marking on it.

In this section, I covered the button components and how they are used in a combination with the Arduino. I also demonstrated some of the less-known quirkiness of the buttons and how to deal with it on both the software and hardware levels. In the next section, I will show you how to use the Arduino to produce sound signals.

Chapter 4 Using Buzzers

Giving visual feedback was always important in electronics. But there are some cases in which the light simply doesn't cut it. A buzzer can be heard even if it's behind the corner or there is an obstacle in between. We humans take audio signals seriously because relatively weak audio signals can even wake a person up from a deep sleep without any problem. It is also said that patients in a deep coma can still hear us clearly. So this sense is definitely important when it comes to electronics.

Audio systems are a chapter of its own and I won't deal with them here. In this example, I'll use relatively inexpensive components called buzzers. This component has been around for decades and it's used mostly for notifying users. By using buzzers, electronics can notify us on some events without us having to constantly watch over some things (like popping popcorn in the microwave, waiting for a customer to enter a store, signaling the end of a game, reminding us that our favorite show is on soon, or simply waking us up).

In this chapter, I'll go through the possible use cases for buzzers. When working with buzzers, please check if it's alright with other people nearby, because buzzers can get really annoying sometimes. Don't abuse the buzzers in your electronic projects, and always have in mind that they can annoy living beings. In short, notify the user with buzzers only when there is something important going on.

Counting Seconds

This is more of a Hello World example when it comes to buzzers. The end result of this example will be something like a clock ticking. Use cases are perhaps a bit limited to measuring a heartbeat, observing a phenomenon, or waiting the right amount of seconds to pass in a short period of time (like, for instance, when blanching veggies). Just imagine all the moments in which you were counting "one Mississippi, two Mississippi" and so on. This is a device that helps you count the seconds.

The most important component in this example is the buzzer. There are many versions available; the one used in this example is depicted in the following picture. But the wiring is pretty much the same regardless of how the buzzer looks like—only two pins are used. The middle pin on this buzzer is left unconnected, like this:

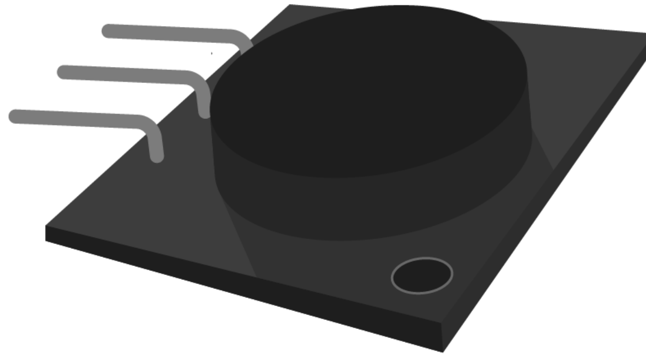


Figure 31: A buzzer

Parts list for this section:

- Arduino Uno
- USB cable
- 1x 100-Ohms resistor for the buzzer
- Breadboard.
- 3x Breadboard jumper wire
- Arduino-compatible passive buzzer

We will start this example with the code. It's similar to the one that we used to make the LEDs blink. The only part that's a bit different is that we'll have to do some math. The pulse for making the sound is rather short and we will make it last just 10 milliseconds long. That's about the right amount of time to perceive a short tick. The duration of this tick has to be taken into account in our loop because, after a while, we would get a significant time drift if we wait for the next cycle for a whole second, without compensating for the duration of the tick.

We would get a drift of around one second per 100 seconds. So the delay in the loop is going to be 990 milliseconds. In the **setup** function, we will register a pin for output and then send short pulses to the pin. After we set up the hardware, it's going to sound something like the old clocks did. Here's the programming part:

```
// We'll use pin 12 for buzzing
int buzzPin = 12;

void setup() {
  // register the pin for output
  pinMode(buzzPin, OUTPUT);
}

void loop() {
  // during ten milliseconds we'll make a tick
  digitalWrite(buzzPin, HIGH);
  delay(10);
```

```
  // then we'll wait 990 millis to tick again
  digitalWrite(buzzPin, LOW);
  delay(990);
}
```

The buzzer can be a little tricky when wiring. It once happened to me that I wired the buzzer the wrong way so I heard a constant sound. When I touched the buzzer, it was extremely hot. So, after you wire the buzzer, check the buzzer to see if it's hot. It could save you the money for the buzzer because, if you don't check it, it could burn out. The good news is that the positive and the negative pin on the buzzer are usually very well marked so you won't have any problems identifying the pins. If your buzzer has more than two pins, it's just fine that one pin on the buzzer is not connected with the buzzer used (in the example, it's the middle one). Here's the wiring for the example:

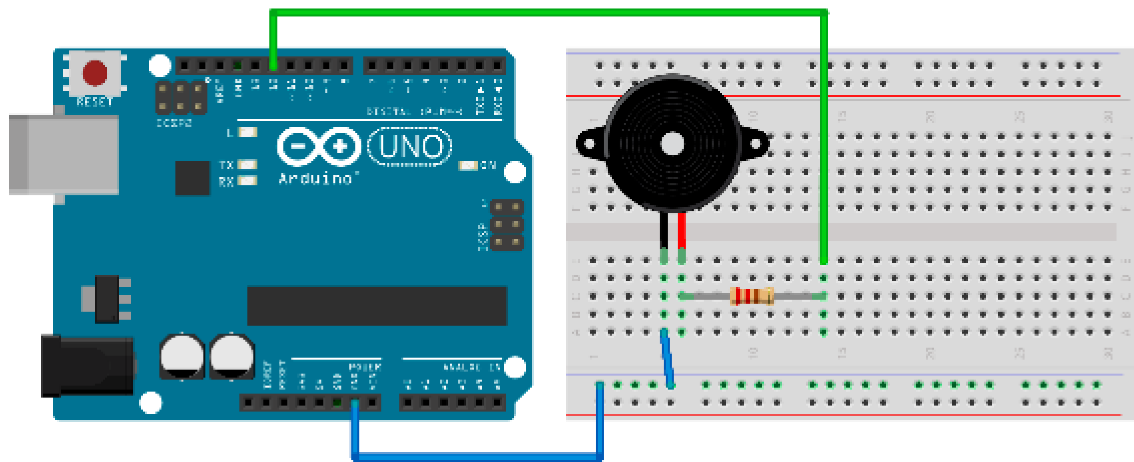


Figure 32: Buzzer counting seconds wiring

All of the wirings to come will be similar to this one. A buzzer's internal resistance is different from one buzzer to another, so you'll have to use the buzzer's data sheet to find the resistance of the buzzer (and then perhaps do some calculations to get the right resistor). In most cases, the 100-Ohm resistor is fine. To calculate the resistor rating, use the data sheet and simply divide five volts by the current rating of your element. If the current rating is 20 mA, then divide five by 0.020 A. This is just an example for you to get the overview of the whole resistor-rating determining process. Also, note that this will give you the overall resistance and you will have to subtract the buzzer's internal resistance from this number. The calculation process is summed up in this note:

Note: Resistor = (5V / Buzzer_current_rating A) – Buzzer_internal_resistance

If you are not sure, the 100-Ohm resistor will cover you in most cases. Arduino is also built so that it's relatively hard to damage it with the most common wiring mistakes because the pins have internal pull-up resistors that limit the current that one pin can draw. I do not consider myself or Syncfusion responsible in any way if you did the following. I connected the buzzer without the resistors and it seemed to work without any problems. Keep in mind that this is a practice with which I do not agree. I did it out of curiosity and to show you that Arduino will cover some of the beginner's mistakes. Now some of you might play around with the buzzer and change the values of the previous program. Let's look at the following code and try to determine what it will do with the buzzer:

```
// what do you think this will do?

// we'll use pin 12 for buzzing
int buzzPin = 12;

void setup() {
  // register the pin for output
  pinMode(buzzPin, OUTPUT);
}

void loop() {
  // we'll keep the pin high for 990 ms
  digitalWrite(buzzPin, HIGH);
  delay(990);

  // and then we'll make just a short break
  digitalWrite(buzzPin, LOW);
  delay(10);
}
```

After running the previous listing, you might be wondering what's going on, and if you somehow typed in the previous example. Don't worry, the example is just fine. Both examples do absolutely the same thing. We have to go a bit into the inner workings of a buzzer to explain what is actually going on here. As a short introduction, I will mention that the sound is actually a vibration traveling through a medium like air or water. And, if you look closely, you will notice that we didn't produce any pulsating currents in the previous example. We are just outputting a current with a constant level on the pin. And what we hear as a tick is a short burst at the beginning of the pulse where the current raises from zero to the `HIGH` value on the pin. We'll go a bit into this in the following section.

Changing Buzzer's Frequency

This section uses completely the same wiring as the previous section does. We'll just change the code as we go along to demonstrate how to change the frequency of a tone generated by the buzzer. Let's go back to the basic principles of a loudspeaker.

A loudspeaker works by vibrating and then pushing the air around, thus causing vibrations that we can hear. There are many loudspeaker designs out there, so I won't go into every single one of them, but I will explain the most basic one instead. In the basic form, a loudspeaker includes a magnet, a vibrating surface, and a coil.

The current goes through the coil and causes the movement of the magnet. If the current starts to alternate, the vibrating surface starts to move back and forth. This then causes the vibration of the air and forms a sound wave. This is the basic functioning principle behind a loudspeaker. Note that, in some speaker designs, the coil is put on the vibrating surface because it is much lighter than the magnet. Actually, this is how most of the speakers are built, but this is an educational example. Here it is depicted with the basic parts:

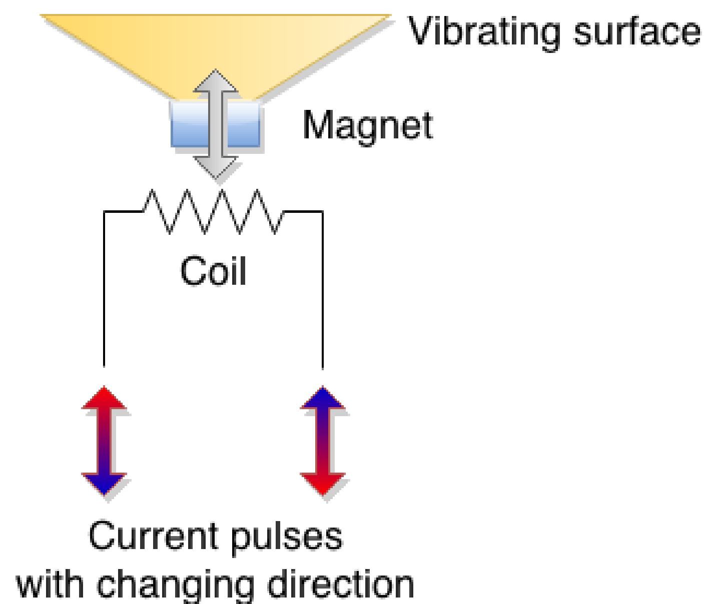


Figure 33: Basic functioning of a loudspeaker

In our previous examples, we didn't send any specific alternating currents to the speaker. We just wrote the **HIGH** value to the pin and then waited while there was a constant current flowing through the pin. There was a constant flow of current; it attracted or repelled the vibrating surface just when it started to flow and then the speaker stabilized. This happened fast and, since there is a mechanical movement involved, it doesn't stop right away. So we heard the vibrations done by the inertia of the vibrating surface. Once per second, we changed the direction of the current and this repeated so we heard a beat. So, yes, in effect, the previous examples were not using the buzzers the way they were supposed to be used. Let's alternate the high and low really fast. Remember, this might annoy the people around you:

```
// we'll use pin 12
int buzzPin = 12;

void setup() {
  // register the pin for output
  pinMode(buzzPin, OUTPUT);
```

```
}

void loop() {
  // high for a short while
  digitalWrite(buzzPin, HIGH);
  delay(1);

  // low for a short while
  digitalWrite(buzzPin, LOW);
  delay(1);
}
```

Now that sounds like something. It's better than the previous ticks; this actually sounds like a tone. Let's have a look at the current:

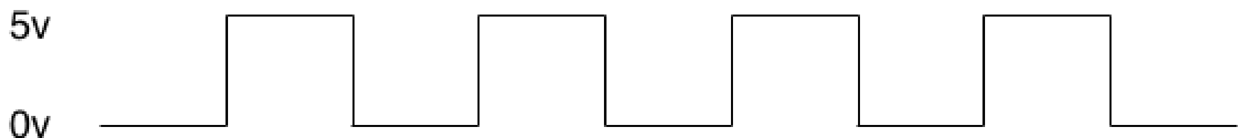


Figure 34: Voltage levels in the previous programming example

Within two milliseconds, we'll go from high to low. We can do that 500 times per second so, in effect, the previous example produces a 500-Hz tone. If you look up the specifications of your buzzer, you might find that it supports frequencies of up to 2,500 Hz or even more. The 500-Hz tone is about as far as we can go with the milliseconds delay. The Arduino also has a **delayMicroseconds** function that can delay execution of a loop for a time shorter than a millisecond.

A microsecond is actually a thousand times smaller than a millisecond. Once again, there are 1,000 microseconds in a millisecond. So, if we wanted to generate a 2,000-Hz tone, this means that we would have to do 2,000 alterations within a second. One alteration would last for a half of a millisecond, and since we have to go from low to high in that time, we'll further divide it by two. This brings us to a delay of a 250 microseconds. The following code will generate a 2,000-Hz tone on the buzzer:

```
// we'll use pin 12
int buzzPin = 12;

void setup() {
  // register the pin for output
  pinMode(buzzPin, OUTPUT);
}

void loop() {
  // high for a short while
  digitalWrite(buzzPin, HIGH);
  delayMicroseconds(250);
```

```
  // low for a short while
  digitalWrite(buzzPin, LOW);
  delayMicroseconds(250);
}
```

Now, we could go on with the various calculations so that we get the frequencies we want. But, if we wanted to do something with the actual tones and program their durations, it would get complicated really fast. Arduino is making all this easier for us by providing the tone function. We'll describe it in the next section.

Using the Tone Function

With the previous example, we had to do some math and write multiple lines of code to produce a tone. To cope with this, Arduino has a built-in **`tone`** function. The function is pretty simple and accepts the number of the pin where the wave will be formed, the frequency of the wave, and the duration. The basic example is shown here; we'll just play a tone of 2,200-Hz:

```
// we'll use pin 12
int buzzPin = 12;

void setup() {
  // register the pin for output
  pinMode(buzzPin, OUTPUT);
}

void loop() {
  tone(buzzPin, 2200, 100);
}
```

The wiring is the same as in the previous examples. If you run this example, you will notice that the tone is constant. The reason for that is that the loop is running really fast and, before the tone is played out, a new tone emitting is started. Let's move the **`tone`** call to the **`setup`** part of the program. Just so that we are sure what's going on, we'll make it last for a whole second:

```
// we'll use pin 12
int buzzPin = 12;

void setup() {
  tone(buzzPin, 2200, 1000);
}

void loop() {
  // leave empty
}
```

The tone function has an interesting property. What do you think the following code will do?:

```
// we'll use pin 12
int buzzPin = 12;

void setup() {
  tone(buzzPin, 2200, 1000);
  tone(buzzPin, 1000, 1000);
  tone(buzzPin, 440, 1000);
}

void loop() {
  // leave empty
}
```

Actually, this is just going to play the last tone, so tone 2,200 and 1,000 won't be heard at all. This is because the calls to the **`tone`** function aren't buffered and, as soon as we call it, the **`tone`** will start to emit. It doesn't matter that we specified a duration of 1,000 milliseconds for the previous call. If we want the tone to get heard, we have to go over other instructions or simply wait. Let's look at what would happen if we put a **`delay`** in between the calls to the **`tone`**:

```
// we'll use pin 12
int buzzPin = 12;

void setup() {
  tone(buzzPin, 2200, 1000);
  delay(2000);
  tone(buzzPin, 1000, 1000);
  delay(2000);
  tone(buzzPin, 440, 1000);
}

void loop() {
  // leave empty
}
```

We can also use **`noTone`** to stop playing the tone on the pin. With the following example, you should hear just a small glitch after the program uploads:

```
// we'll use pin 12
int buzzPin = 12;

void setup() {
  tone(buzzPin, 2200, 1000);
  noTone(buzzPin);
}
```

```
void loop() {
  // leave empty
}
```

With this, we covered the use cases of the **tone** function. With the examples I showed you up until now, you could make nice warning sounds or something similar. But that's not all there is to buzzers; a lot of people are using Arduino and a buzzer to play a melody. I'll go over how to play a melody on a buzzer in the next section.

Playing a Melody

Up until now, we covered the theoretical part of making sounds with Arduino. But some of the readers of this book will have some kind of musical education and, hopefully, they will enjoy this section. Remember, we'll keep it short and to the point.

You don't have to worry about the wiring as it's the same all along. Just as a brief introduction, a note has a frequency, and music combines the frequencies with the durations. The buzzer cannot actually reproduce frequencies lower than 31 Hz. We mentioned earlier that most of the buzzers out there don't go over 2,500 Hz. We'll look at how to play a melody in the following code listing. The initial part of the listing is just going to provide you with the list of notes and their frequencies. You can reuse them to play your own melodies:

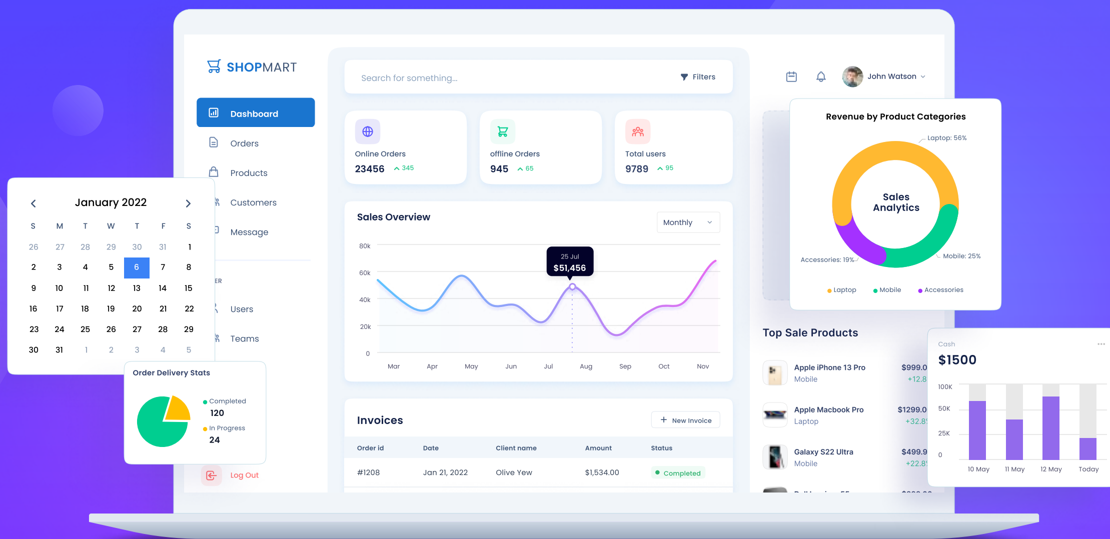
```
#define _SILENCE 0
// most of the buzzers don't work with very low tones
#define _B0  31
#define _C1  33
#define _CS1 35
#define _D1  37
#define _DS1 39
#define _E1  41
#define _F1  44
#define _FS1 46
#define _G1  49
#define _GS1 52
#define _A1  55
#define _AS1 58
#define _B1  62
#define _C2  65
#define _CS2 69
#define _D2  73
#define _DS2 78
#define _E2  82
#define _F2  87
#define _FS2 93
#define _G2  98
#define _GS2 104
#define _A2  110
#define _AS2 117
#define _B2  123
#define _C3  131
```

```
#define _CS3 139
#define _D3  147
#define _DS3 156
#define _E3  165
#define _F3  175
#define _FS3 185
#define _G3  196
#define _GS3 208
#define _A3  220
#define _AS3 233
#define _B3  247
#define _C4  262
#define _CS4 277
#define _D4  294
#define _DS4 311
#define _E4  330
#define _F4  349
#define _FS4 370
#define _G4  392
#define _GS4 415
#define _A4  440
#define _AS4 466
#define _B4  494
#define _C5  523
#define _CS5 554
#define _D5  587
#define _DS5 622
#define _E5  659
#define _F5  698
#define _FS5 740
#define _G5  784
#define _GS5 831
#define _A5  880
#define _AS5 932
#define _B5  988
#define _C6  1047
#define _CS6 1109
#define _D6  1175
#define _DS6 1245
#define _E6  1319
#define _F6  1397
// some buzzers don't produce tones before 1500 Hz
#define _FS6 1480
#define _G6  1568
#define _GS6 1661
#define _A6  1760
#define _AS6 1865
#define _B6  1976
#define _C7  2093
#define _CS7 2217
#define _D7  2349
// only higher quality buzzers after 2500 Hz
#define _DS7 2489
#define _E7  2637
```

```
#define _F7  2794
#define _FS7 2960
#define _G7  3136
#define _GS7 3322
#define _A7  3520
#define _AS7 3729
#define _B7  3951
#define _C8  4186
#define _CS8 4435
#define _D8  4699
#define _DS8 4978

int buzzPin = 12;

// predefined melody
int melody[] = {
  _C4, _G3, _G3, _A3, _G3, _SILENCE, _B3, _C4};

#define melodySize (sizeof(melody)/sizeof(int *))

// durations: 4 = quarter, 8 = eighth, etc.:
int noteDurations[] = {
  4, 8, 8, 4, 4, 4, 4, 4 };

void setup() {
  // going over the notes in the melody:
  for (int noteIndex = 0; noteIndex < melodySize; noteIndex++) {

    // note duration = one second / duration
    // quarter note = 1000 / 4, eighth note = 1000/8 ...
    int noteDuration = 1000/noteDurations[noteIndex];
    tone(buzzPin, melody[noteIndex], noteDuration);

    // let's wait a little bit longer so that we can hear the tones
    int pauseBetweenNotes = noteDuration * 1.10;
    delay(pauseBetweenNotes);
    // stop the tone and start with the next
    noTone(buzzPin);
  }
}

void loop() {
  // leave empty
}
```

As played with the melody in the previous example, we have reached the final section regarding buzzers. We went from the basic wiring of a buzzer to the ways in which to use it from the software side. There are a lot of applications for giving audio feedback to the users of Arduino-based projects, and the provided examples will help you get started when working with the buzzers, whether it is notifying the user or playing a melody. In the next chapter, we are going to introduce how to measure environment conditions.

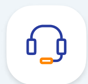
THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING POWERFUL APPS

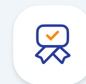


GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

1,700+ components for mobile, web, and desktop platforms


Support within 24 hours on all business days


Uncompromising quality

Hassle-free licensing

28000+ customers

20+ years in business

Trusted by the world's leading companies

Chapter 5 Measuring Environment Conditions

By now, the only input to the Arduino that we covered is the reacting to a press of a button. This is a relatively simple reading in which the current is either present or not. Based upon the current presence, we can then do various actions. Arduino is capable of far more than just detecting whether or not a button is pressed; it can be used to measure the environment conditions. Environment measurements are rarely simple binary values. For instance, the light level in the room might be somewhere between totally dark and very bright from the sun coming in from a glass ceiling. For instance, we could turn on the light if the light level drops below a predefined value. In this section, we will explain how to measure the air temperature, air humidity, and the barometric pressure as well as how to detect the light levels. As in the previous chapters, we will use widely available and inexpensive components to make the examples. Let's start with simple air temperature readings.

Measuring Air Temperature

We'll start by using a simple electronic component called the Thermistor. The basic principle behind this component is that it gives different resistance to the passing current based upon the surrounding temperature. There are two basic types of Thermistors:

- Negative Temperature Coefficient - Resistance drops as temperature increases
- Positive Temperature Coefficient - Resistance increases as temperature increases

In our example, we'll use the Negative Temperature Coefficient type. The 10K Ohm NTC 5mm Thermistor is a good choice when it comes down to availability, precision, and price:

Figure 35: 10K Ohm NTC 5mm Thermistor

Parts list for this section:

- Arduino Uno
- USB cable
- 1x 10K Ohm NTC 5mm Thermistor
- 1x 10k Ohm resistor

- Breadboard
- 3x Breadboard jumper wire

The behavior of the Thermistor is determined by the Steinhart-Hart equation. The most important parameter in the equation is the measured resistance. The measured resistance is not a digital value of **HIGH** and **LOW**. It has a whole range of values between the **HIGH** and the **LOW**. Arduino uses analogue pins to measure those values. With the binary pins, we only get the information if the current is present on some pin or not. With the analogue, we get a value from a range. In Arduino, this range is from 0 to 1023—meaning that there are 10 bits to represent the input current on the pin. Analogue pins can also be used for output, but then they behave as regular digital pins, and it is not possible to read analogue values from them. Most of the time, you won't have problems with this behavior, but keep in mind that it could happen. Let's start with the wiring for the example:

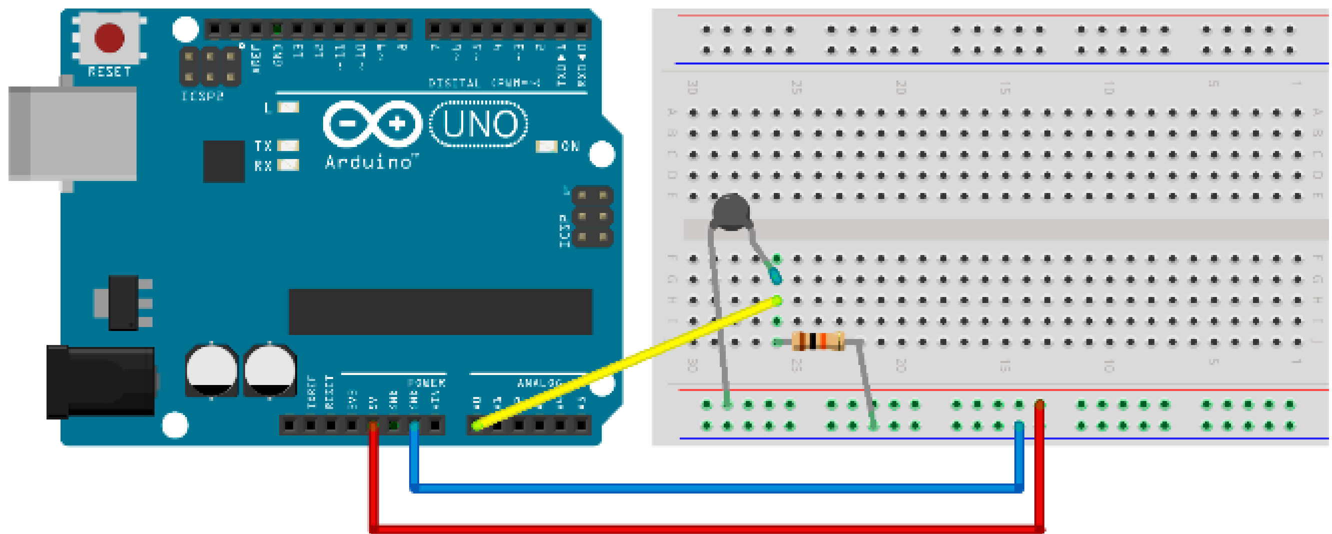


Figure 36: Wiring for 10K Ohm Thermistor

The previous example uses just three wires instead of five. Replace the longer pins of elements with wires if you desire. It's the usual way of wiring in the literature and schematics but, for smaller circuits, it's perfectly fine not to use too many wires. We'll output the readings to the Serial port. The most complicated part of this example is reading the input current and then turning this reading into the temperature. Most of the examples found online use a couple of one-liners and then magically show the output to the user. You'll learn more by understanding the code than by just copying and pasting the example. So we are going to go along with this implementation:

```
// we need advanced math functions
#include <math.h>

int reading;

void setup() {
```

```
  // Initialize serial communication at 9600 bit/s
  Serial.begin(9600);
}

void loop() {
  reading = analogRead(A0);

  float kelvins = thermisterKelvin(reading);
  float celsius = kelvinToCelsius(kelvins);
  float fahrenheit = celsiusToFahrenheit(celsius);

  Serial.print("kelvins = ");
  Serial.println(kelvins);

  Serial.print("celsius = ");
  Serial.println(celsius);

  Serial.print("fahrenheit = ");
  Serial.println(fahrenheit);

  Serial.println("");

  delay(3000);
}

float thermisterKelvin(int rawADC) {
  // Steinhart-Hart equation
  // 1/T = A + B (LnR) + C(LnR)ˆ3

  // LnR is the natural log of measured resistance
  // A, B, and C are constants specific to resistor

  // we'll use:
  // T = 1 / (A + B(LnR) + C(LNR)ˆ3)

  // we are using long because Arduino int goes up to 32 767
  // if we multiply it by 1024, we won't get correct values
  long resistorType = 10000;

  // input pin gives values from 0 - 1024
  float r = (1024 * resistorType / rawADC) - resistorType;

  // let' calculate natural log of resistance
  float lnR = log(r);

  // constants specific to a 10K Thermistor
  float a = 0.001129148;
  float b = 0.000234125;
  float c = 0.0000000876741;

  // the resulting temperature is in kelvins
  float temperature = 1 / (a + b * lnR + c * pow(lnR, 3));

  return temperature;
```

```
}

float kelvinToCelsius(float tempKelvin) {
  return tempKelvin - 273.15;
}

float celsiusToFahrenheit(float tempCelsius) {
  return (tempCelsius * 9.0)/ 5.0 + 32.0;
}
```

After running the example, wrap your fingers around the thermistor and hold it for a while. Your body temperature should be higher than the air's temperature, so you'll see the change in the readings. We used the Serial Monitor in the previous examples.

Detecting Light Levels

Light level detection is pretty similar to temperature measuring. The component that we will use in this section becomes more conductive with the light shining upon it:

Figure 37: Photo Resistor

Parts list for this section:

- Arduino Uno
- USB cable
- 1x GL5528 Photo Resistor or any 10K Ohm Photo Resistor
- 1x 10k Ohm resistor
- Breadboard
- 3x Breadboard jumper wire

The wiring for the example looks similar to the one for measuring the air temperature:

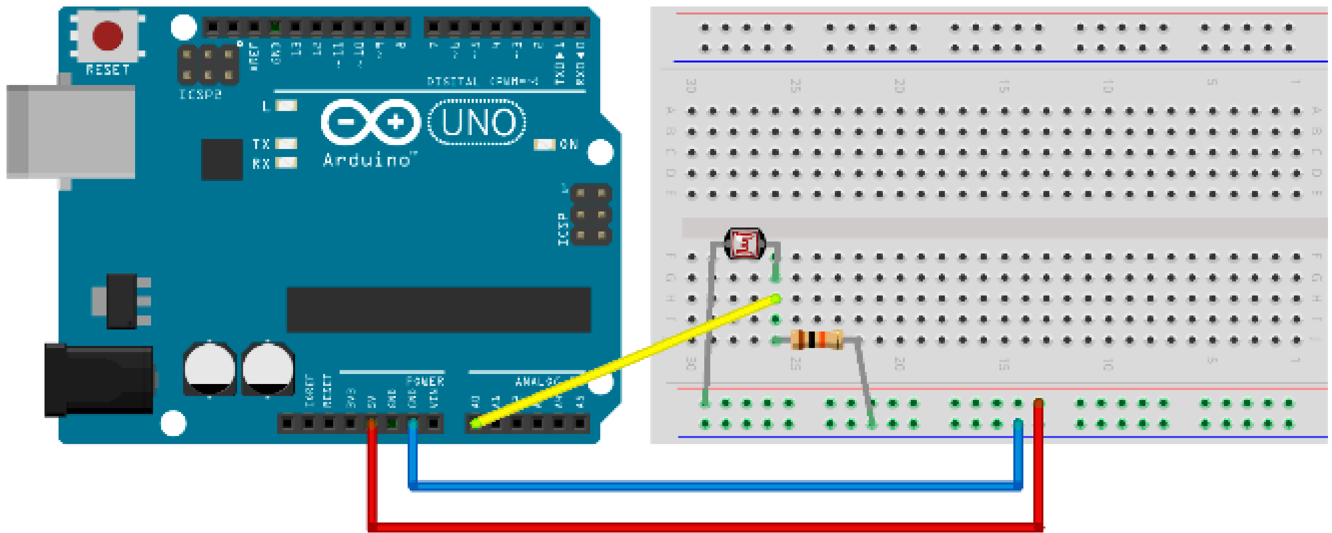


Figure 38: Wiring a photo resistor

Let’s just skip to code right away and explain it later:

```
// we'll store reading value here
int reading;

void setup() {
  // initialize serial communication at 9600 bit/s
  Serial.begin(9600);
}

void loop() {
  reading = analogRead(A0);

  // we'll measure the input values and map 0-1023 input
  // to 0 - 100 values
  int lightLevel = map(reading, 0, 1023, 0, 100);

  Serial.print("light = ");
  Serial.println(lightLevel);

  delay(3000);
}
```

The first question that might come to mind is what does this number mean? It goes from 0 to 100, so one might even think of it as a percentage. But the question is, compared to what? If the photo resistor is connected in a circuit (as in the previous figure), it can measure light levels of up to 10 lux. Lux is a measurement for luminance. 10 lux is not a lot if you look at the following table; actually, it’s pretty dark and you can’t measure the differences between bright and brighter:

Table 1: Lux levels and a description of the light levels

Lux Level	Description
0.27 – 1.0	Moonlight
3.4	Dusk
10 lux	Very dark and rainy day
50 lux	Regular light level in a living room
320 – 500	Office light
1000	Full daylight with no direct sun
100 000	Sunlight

In our example, we are not measuring light levels in lux units but, rather, are simply mapping the reading to a percentage. Percentages are easily understandable, and it's easy to check on which light levels we want to take action on. However, with the previous setting, we won't be able to measure very bright light. There's a neat trick to get around this situation. If you put a 1k Ohm resistor into the circuit instead of the 10, you will be able to get differences in readings on a much higher level, and you will be able to differentiate between the bright and brighter.

If you want to measure the exact lux levels, you will have to carefully look into the resistor specification and map the voltages from the specification to your readings. This work is a bit tedious, and it's used only in very specific situations such as photography. For the regular use case such as "Is there a light on?" or something similar, it's advisable that you simply use the percentages and adapt the resistor in the circuit to the light levels that you want to measure. Basically, use a 10K resistor if you want to differentiate between the dark and darker, or a 1K resistor if you want to differentiate between the light and lighter. Most readers with an engineering background might prefer the lux unit, but mapping the voltage to simple percentages is more than sufficient for day-to-day use.

Using Temperature and Humidity Sensor

In the earlier section, we used an element called a thermistor to measure the temperature. The electronics behind it were relatively simple: There's an element and we're just interpreting the current levels that we get. For some usages, having a simple element just doesn't cut it, and using the basic elements significantly increases the number of elements that we have to put into our electronics. So the manufacturers started to build separate components that can then communicate with the Arduino, which then communicates with other devices, and so forth. We will start with something relatively simple such as measuring the temperature in order to go through the whole process of using a separate component with the Arduino.

There are many temperature sensors that can communicate with the Arduino and exchange information with it. One of the most commonly used and relatively inexpensive is the DHT11 sensor. Besides the temperature, it can measure the relative humidity. Programming it might be a bit tricky, and we would have to go deep into the specification to do any work with it. So we are going to use a library to do the dirty work for us. Here's the DHT11 module:

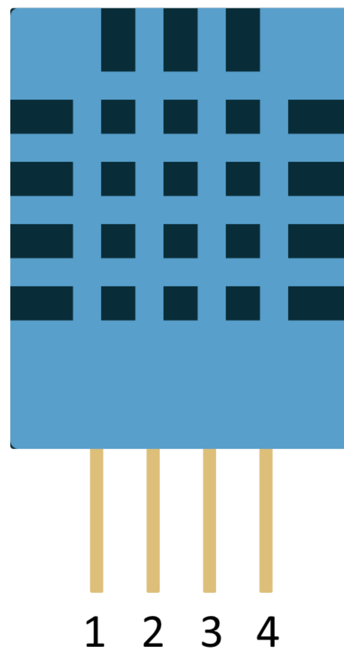


Figure 39: DHT11 temperature and humidity sensor

The figure shows markings on the pins so that it'll be easier for you to connect the sensor to the board when you start setting up the example. You don't have to worry about turning the sensor the wrong way around because the front side shown in the figure has holes whereas the back side is smooth. As in every section, we'll start with the parts.

Parts list:

- Arduino Uno
- USB cable

- DHT11 temperature and humidity sensor
- 10K Ohm resistor
- Breadboard
- 5x Breadboard jumper wire

And here's the wiring:

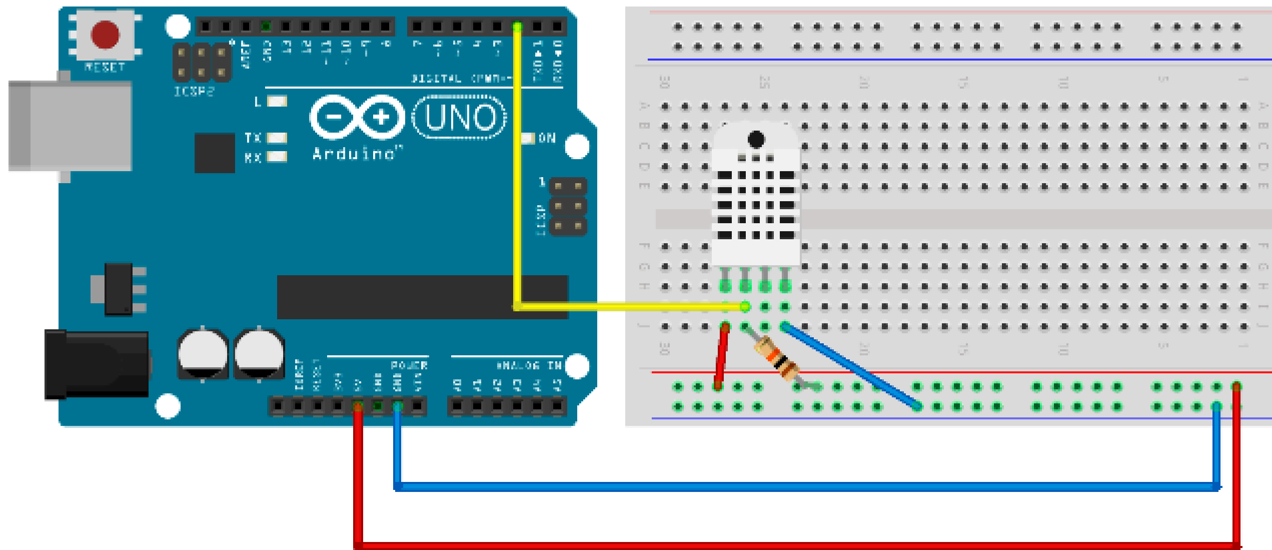


Figure 40: DHT22 (depicted) and the DHT11 have same wiring

There are variations of the DHT, such as DHT22. We depicted the wiring with the DHT22 just so you are aware that there are other sensors with almost the same name but they function totally different (although the wiring is the same). We'll look at that when we discuss programming.

To hook everything up, we'll need to download a library. Download the file located in Github here and remember where you saved it. Create a new sketch and after that, click **`Sketch`** > **`Include Library`** > **`Add ZIP Library`**. Select the zip archive that you downloaded and the IDE should automatically install the library. Here's the code that goes along with the example. Make sure you change the DHTTYPE constant to the type you're using:

```
#include "DHT.h"

// to use the library we have to define some constants

// what pin are we going to use
#define DHTPIN 2

// what type of sensor are we going to use
// possible: DHT11, DHT22, DHT21
// be careful to set it correctly
#define DHTTYPE DHT11
```

```
// instantiate DHT sensor
DHT dht(DHTPIN, DHTTYPE);

void setup() {
  // we'll print measurements to serial
  Serial.begin(9600);

  // start the sensor
  dht.begin();
}

void loop() {
  // it takes around 2 seconds for the sensor to update data
  delay(2000);

  // read humidity
  float h = dht.readHumidity();

  // Read temperature as Celsius
  float t = dht.readTemperature();

  // Read temperature as Fahrenheit
  float f = dht.readTemperature(true);

  // check the value
  if (isnan(h) || isnan(t) || isnan(f)) {
    Serial.println("Failed to read, reading again!");
  }
  else {
    // compute heat index
    // heat index works only with Fahrenheit
    float hi = dht.computeHeatIndex(f, h);

    Serial.print("Humidity %: ");
    Serial.println(h);

    Serial.print("Temperature *C: ");
    Serial.println(t);

    Serial.print("Heat index *C: ");
    Serial.println(fahrenheitToCelsius(hi));

    Serial.print("Temperature *F: ");
    Serial.println(f);

    Serial.print("Heat index *F: ");
    Serial.println(hi);


    Serial.println();
  }
}

float fahrenheitToCelsius(float tempF) {
```

```
  return (tempF - 32) / 1.8;
}
```

Many Arduino sensors work similarly to this one. You download a library, hook up the sensor, and then start reading the data. There are many libraries out there for various Arduino sensors. This section described the standard procedure when working with more complex sensors.

Measuring Barometric Pressure

This is not a book on physics, so we won't talk a lot about what air pressure is. Air also has weight, and surfaces exposed to the atmosphere experience pressure of the air. There are two main ways to use barometric pressure. One way is to track changes in the weather. A second way is to measure the change in the altitude between two points with a known pressure.

When observing the weather, there are a lot of weather stations located either very high in the mountains, at the sea level, or below it (like in Death Valley). No matter where we take the measurement, we can calculate what the pressure would be like on the sea level if we know the height above the sea level. So, in the weather forecast, all of the air pressure values are usually expressed at the sea level. The standard pressure is 1013.25 millibar or 29.92 inHg. We can then quickly determine if a certain point is below the usual pressure value or above it. Depending upon the location we are currently on, we can even use the movements of the air pressure to predict the weather.

Air gets thinner when it goes higher into the atmosphere. Today, sensors can measure relatively small vertical movements; usually the precision is within three feet or one meter. The main component we will use in this section is the BMP180 Barometric Pressure sensor:

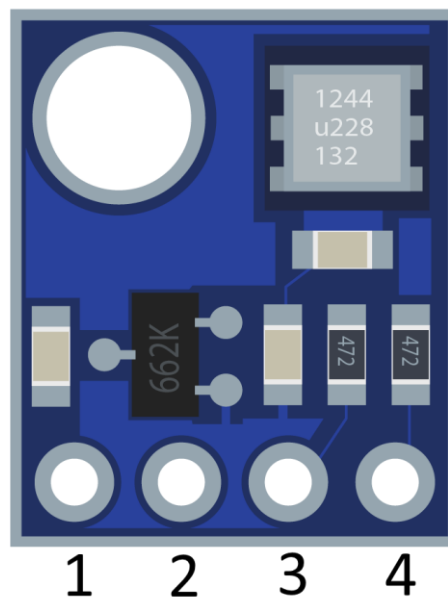


Figure 41: BMP180 barometric pressure sensor

Parts list for this section:

- Arduino Uno
- USB cable
- BMP180 barometric pressure sensor
- Breadboard
- 4x Breadboard jumper wire

The external pin goes to a 3.3V power source on Arduino. Pin 2 goes to the ground. Pin 3 goes to A5 and Pin 4 goes to A4. Note that there are varieties of the chip and that some of them have five pins or more. In a five-pin version, the first pin is simply ignored and the rest is the same as in the previous figure. If there are more than five pins, look up the manufacturer's documentation. Let's look at the wiring:

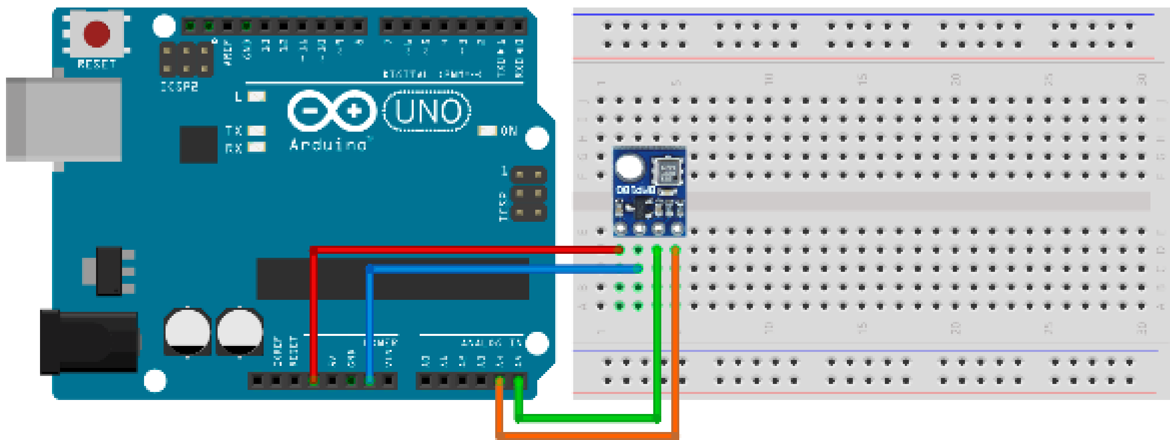


Figure 42: BMP180 sensor wiring

Reading the value is a bit more complicated. Communication with the module would be a labor-intensive process. So to hook everything up, we'll need to download a library. Download the file located on Github here and remember where you saved it. Create a new sketch and click **`Sketch`** > **`Include Library`** > **`Add ZIP Library`**. Select the zip archive that you downloaded and the IDE should automatically install the library. Here's the code that goes along with the example:

```
#include <SFE_BMP180.h>
// SFE_BMP180 library is not enough
// we have to include the wire to communicate with the sensor
#include <Wire.h>

// SFE_BMP180 object sensor:
SFE_BMP180 sensor;

// altitude of my home in meters
// change this to your own altitude!!!
// you will get strange reading otherwise
```

```
#define ALTITUDE 178.0

// we'll keep track of previous measured pressure
// so that we can calculate relative distance
double previous_p = 0;

void setup() {
  Serial.begin(9600);
  Serial.println("INIT ...");

  // begin initializes the sensor and calculates the calibration values
  if (sensor.begin()) {
    Serial.println("BMP180 init OK");
  }
  else {
    Serial.println("BMP180 init FAILED!!!");
    // stop the execution
    delay(1000);
    exit(0);
  }
}

void loop() {
  // we'll make a reading every 5 seconds
  delay(5000);

  byte waitMillis, status;
  double temp, p, p0, a;

  // weather reports use sea level compensated pressure
  // find out your altitude to get the readings right

  Serial.println();
  Serial.print("ALTITUDE: ");
  Serial.print(ALTITUDE);
  Serial.print(" m  ");
  Serial.print(ALTITUDE * 3.28084);
  Serial.println(" feet");

  // to read the pressure, you must read the temperature first

  // ask the sensor when can we get a temperature reading
  // if o.k. we get waitMillis, otherwise 0

  waitMillis = sensor.startTemperature();
  if (waitMillis == 0) {
    Serial.println("Temperature reading not ready");
    return;
  }

  // wait for as long as the sensor says
  delay(waitMillis);

  // retrieve the temperature measurement into temp
```

```
// function updates waitMillis again
waitMillis = sensor.getTemperature(temp);

if (waitMillis == 0) {
  Serial.println("Error reading temperature");
  return;
}

// current temperature values in different units
Serial.print("TEMPERATURE: ");
Serial.print(temp);
Serial.print(" *C  ");
Serial.print((9.0 / 5.0) * temp + 32.0);
Serial.println(" *F");

// start a pressure measurement:
// precision from 0 to 3, precise readings take more time
// we get millis how long to wait for the measurement

waitMillis = sensor.startPressure(3);

if (waitMillis == 0) {
  Serial.println("Error starting pressure reading");
  return;
}

delay(waitMillis);
// Retrieve the completed pressure measurement:
// Function returns 1 if successful, 0 if failure.

status = sensor.getPressure(p,temp);
if (waitMillis == 0) {
  Serial.println("Error getting pressure reading");
  return;
}

// pressure data
Serial.print("ABSOLUTE PRESSURE: ");
Serial.print(p);
Serial.print(" mb  ");
Serial.print(p * 0.0295333727);
Serial.println(" inHg");

// sensor returns absolute pressure, which varies with altitude.
// in weather reports a relative sea-level pressure is used
p0 = sensor.sealevel(p, ALTITUDE);

Serial.print("SEA LEVEL: ");
Serial.print(p0);
Serial.print(" mb  ");
Serial.print(p0 * 0.0295333727);
Serial.println(" inHg");

// altitude difference from previous reading
```

```
  a = sensor.altitude(p, previous_p);
  Serial.print("HEIGHT DIFFERENCE FROM PREVIOUS: ");
  Serial.print(a);
  Serial.print(" m  ");
  Serial.print(a * 3.28084);
  Serial.println(" feet");

  previous_p = p;
}
```

The example might give you significantly different readings than the one on the local weather station. Make sure that you find out the sea level height of your home before you run the example. Otherwise, you might be confused by the results. The height is set by using the define command **`#define ALTITUDE 178.0`** in line 12 of the provided example. If you are ever going to build a project that does weather monitoring, this will be one of examples that will come in handy. By now, we talked a lot about the atmosphere. What else can we sense with the Arduino? We'll talk about it more in the next section.

Detecting Soil Moisture

If you ever had plants, you know that one can occasionally forget to water them. Well, believe it or not, Arduino has this covered too. All you need to avoid that kind of problem is a soil moisture sensor. Soil conductivity properties change with the amount of water contained in the soil, and the sensor measures those properties. The sensor has two components: the probe and the sensor itself.

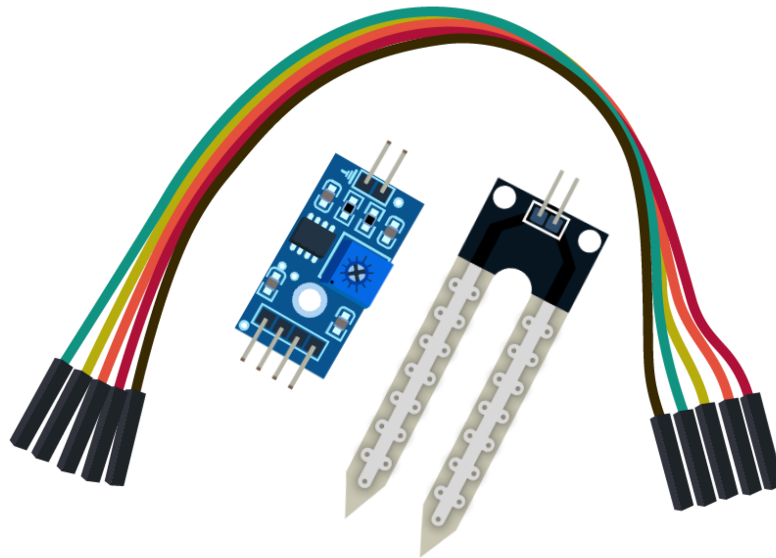


Figure 43: Soil moisture sensor with parts

The wiring between the probe and the sensor is done with two wires. It doesn't matter how you connect them. We'll talk about the wiring between the Arduino and the sensor shortly. We won't use the breadboard in this example.

Parts list for this section:

- Arduino Uno
- USB cable
- YL-96 soil moisture sensor
- 4x wire
- A glass of water to test the functioning

Table 2: YL-96 pins

Pin	Description
VCC	Pin for connecting 5V +
GND	Pin for connecting the ground
DO	Digital output
AO	Analog output

Connect the VCC to Arduino 5V, GND to Arduino GND, and DO to digital in 8. Use the following code listing:

```
// we'll use digital pin 8
int inPin = 8;

// we'll store reading here
int val;

void setup() {
  // initialize serial communication
  Serial.begin(9600);
  // set the pin mode for input
  pinMode(inPin, INPUT);
}

void loop() {
  // read the value
  val = digitalRead(inPin);

  // if the value is low
  if (val == LOW) {
```

```
    // notify about the detected moisture
    Serial.println("Moisture detected!");
    delay(1000);
  }
}
```

Open the Serial Monitor and watch what happens when you put the probe in the glass of water. Remember, this is just a digital reading. The probe can use the analog pins to send the levels of moisture. Reconnect the pins and connect the VCC to Arduino 5V, the GND to Arduino GND, and the AO to Analog in 5 on the Arduino. Use the following code sample:

```
// we'll store reading from the input here
int input;

// we'll map the input to the percentages here
int val;

void setup() {
  // initialize the serial communication
  Serial.begin(9600);
}

void loop() {
  // read the value from A5
  input = analogRead(A5);
  // map it to percentage
  val = map(input, 1023, 0, 0, 100);

  // print and wait for a second
  Serial.println(val);
  delay(1000);
}
```

Open the Serial Monitor and check what's going on if you put the probe into a glass of water. Now, you might ask, "Why is the reading only at around 40 percent when, if it's dipped into water, it should be around 100 percent? Well, moist soil is actually a lot better than a glass of water at conducting electricity. If you have a chance, try putting the sensor into moist soil (but be careful not to make a mess with the soil grains).

Also, be very careful with the glass of water because, if you spill it, it may damage the Arduino or nearby electronic devices. Now, you might think that that's all there is to it, but there's a very important catch with this sensor. The probe starts to corrode after a while and, within a couple of days (or maybe weeks), it will be completely unusable.

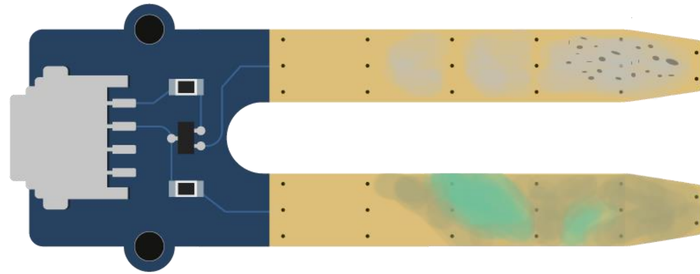


Figure 44: Corroded soil moisture sensor

We will go around the sensor corrosion by giving the sensor electricity only when it will actually need to measure the moisture. Sampling moisture is usually not done every second; once per minute or hour is perfectly fine. Let's rewire the sensor so that it supports being on only when we make the measurement. Connect the VCC to the Arduino digital pin 8, the GND to Arduino GND, and the AO to Analog in 5 on the Arduino. Use the following code to turn on the sensor once per minute and then make a reading:

```
// we'll store reading from the input here
int input;

int vccPin = 8;

// we'll map the input to the percentages here
int val;

void setup() {
  // initialize the serial communication
  Serial.begin(9600);

  // set up the pin mode
  pinMode(vccPin, OUTPUT);
}

void loop() {
  // turn on the sensor
  digitalWrite(vccPin, HIGH);

  // wait a little bit for the sensor to init
  delay(100);

  // read the value from the A5
  input = analogRead(A5);

  // turn off the sensor
  digitalWrite(vccPin, LOW);

  // map it to percentage
  val = map(input, 1023, 0, 0, 100);
```

```
  // print and wait for a minute
  Serial.println(val);

  delay(60000);
}
```

Using this code might save you a lot of probes because they go out fast if exposed to constant current. Many people that start to use Arduino in gardening are often surprised by this effect, but this is actually a known issue when using Arduino for gardening. Also, there are more expensive sensors available that can last significantly longer, but the solution where an inexpensive sensor is used with a little bit of smart code is what Arduino is all about.

Measuring Environment Conditions Conclusion

This is a very important chapter to many people. Measuring the environment conditions is one of the top reasons why they even start with the Arduino in the first place. People simply want to know what the temperature in the room is while they are away, or are simply interested what's going on in environments when they are not around.

Most of those concerns are related to the energy costs, environmental concerns, and savings. The heating does not have to be on all of the time. By using Arduino in combination with some sensors, we can look at the historic data or chart them to a graph, and quickly determine where potential savings could be made.

There are also people with health issues and they want to specifically target an exact level of humidity in their environment; Arduino can be of great help there, too. One of the main advantages of Arduino as compared to ready-made solutions is that you can always tweak it to match only your needs—without having to acquire expensive devices when you only need just a part of their functionality.

Also, there are more and more people who want to eat healthier, and they are starting to grow their own food (well, perhaps not all of their food but at least some). There is a significant movement called Urban Gardening (UG) and Arduino is definitely a go-to technology in many UG-related applications. But the possibilities of Arduino do not end with just the environmental conditions. Actually, Arduino can be used to do a lot more. We will cover it in the next chapter.

Chapter 6 Detecting Objects

By now, we learned how to monitor environment properties. But, often we will want to react to a physically changing environment or other objects moving or performing actions around us. That kind of sensing electronics is already used in everyday life.

For instance, if you ever drove a car backward and had parking sensors, you probably were already using object detection technology. Another popular technology for sensing is the intruder detection in various alarm systems. We will describe how those systems work, too.

Detecting objects is often not a binary business, and we will want a level of configuration. So, in this section, we will start with something simple that does not have a direct and obvious connection with the sensor for detecting objects. But it's actually very important because we won't always want the detecting components to react if the values pass or fall below a predefined value. But we will want to be able to define the sensitivity level, without having to reprogram the whole Arduino project. The component that enables us to make variable settings is called a potentiometer. Basically, what it does is change its electrical resistance when we turn a knob around. We will use a potentiometer to change sensitivity level in later examples. Let's start with the potentiometer.

Using Potentiometers

Potentiometers usually come in the form of a turning knob. This knob provides more resistance the more it is turned. It is usually used to control the volume on radios, fan speeds on ventilation systems, or the desired temperature level in rooms. You probably have used it by now at least once in your life, but let's look at how it works and how we can put it to use.

Parts list for this section:

- Arduino Uno
- USB cable
- 10K Ohm potentiometer
- Breadboard
- 3x Breadboard jumper wire

The wiring for the example is relatively simple. Keep in mind that it doesn't really matter how you connect the positive and negative pins to the potentiometer. The only thing that will change if you make different wiring for the positive and the negative pins is that the resistance will change from a smaller to a larger value when you turn the knob, or the other way around. It all depends upon how you connected it. Let's look at the wiring:

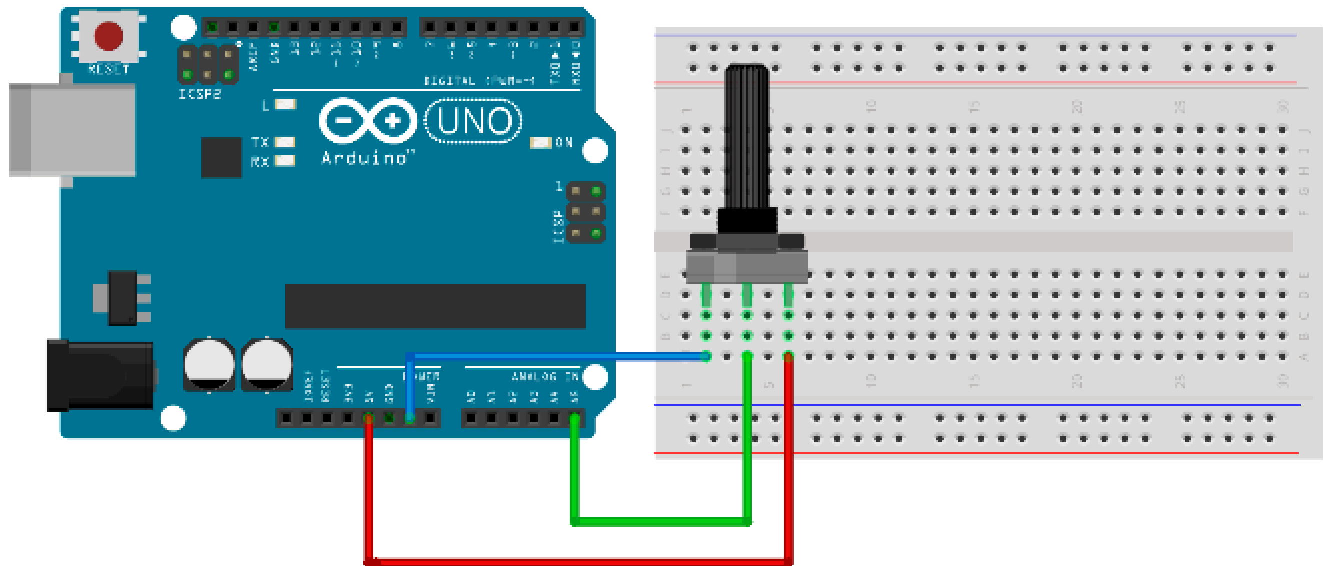


Figure 45: Potentiometer wiring

Use this programming example to see what's going on:

```
// we'll store reading from the input here
int input;

// we'll map the input to the percentages here
int val;

// for keeping the previous reading
int previous_val;

void setup() {
  // initialize the serial communication
  Serial.begin(9600);
}

void loop() {
  // read the value from A5
  input = analogRead(A5);

  // map it to percentage
  val = map(input, 1023, 0, 0, 100);

  // if percentage changed
  if (val != previous_val) {
    // print and store the value for comparison
    Serial.println(val);
    previous_val = val;
  }
```

```
  // delay to prevent output overcrowding
  delay(100);
}
```

If you turn the knob, you should see how the percentage changes from 0 to 100. If you don't like the direction the value rises or falls, you have two options. One option is to exchange the rightmost and the leftmost connectors as previously mentioned. Try it out and see how the knob behaves, and then reconnect it so that we can see how to do it with the software:

```
  val = map(input, 1023, 0, 100, 0);
```

It is really up to you how you will adapt the reading values to the knob. The good thing about the analog components is that you instantly get the feedback no matter where and how you use them, so you can quickly compensate if you went in the wrong direction. Let's start with another building block that is often used when detecting objects in the environment and especially for measuring the distance. Depending upon the Arduino build, it might disconnect or stop sending data in the minimum and maximum potentiometer rotation points. If that happens, disconnect the Arduino and reconnect it.

Using Ultrasonic Distance Sensor

Ultrasonic distance sensors are a very inexpensive and practical way to check how far something is. Sound travels through the atmosphere at a known speed. The speed can vary depending upon the weather conditions but, according to Wikipedia, sound passes a mile in 4.689 seconds or a kilometer in 2.914 seconds. The basic working principle behind the sensor is that we simply measure the time it takes between emitting an ultrasonic sound wave and then waiting for it to come back. Then, we divide the time by two because the wave traveled double the distance. Then we simply divide this duration by the speed of sound. The basic functioning principle is shown in the following figure:

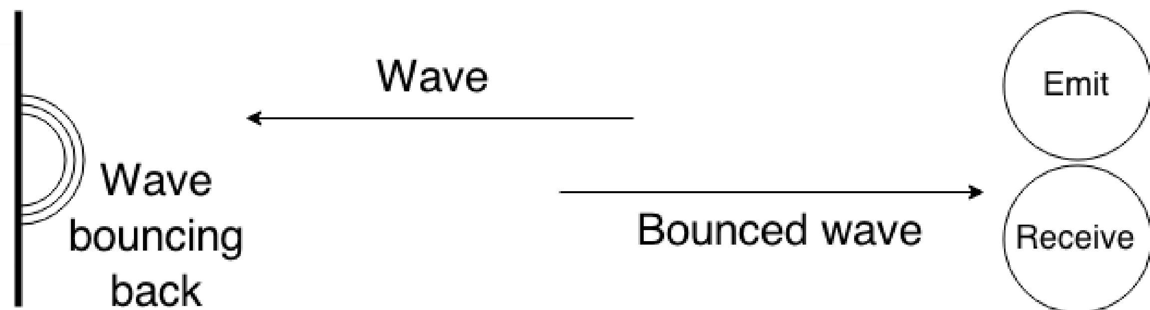


Figure 46: Ultrasonic distance sensor basic working principle

The previous figure shows two components. One is for emitting and the other is for receiving the sound wave. The ultrasonic distance sensor looks similar to the two circles displayed. It's shown in the following figure:

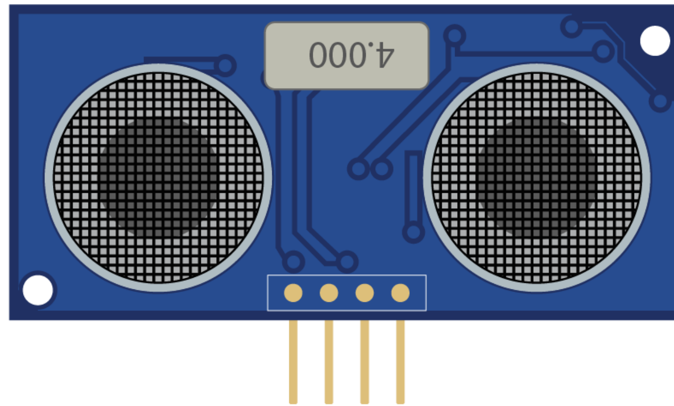


Figure 47: Ultrasonic distance sensor

The module used in our case is very accessible and inexpensive; its designation is HC-SR04 and it is in the parts list for this section.

Parts list for this section:

- Arduino Uno
- USB cable
- HC-SR04 Ultrasonic distance sensor
- Breadboard
- 4x Breadboard jumper wire

The basic overview of the wiring is show here:

Table 3: HC-SR04 Pins and Connections to Arduino

Pin	Connection
Vcc	Pin for connecting 5V +
Trig	Connect to pin 13 in our example
Echo	Connect to pin 12 in our example

Pin	Connection
Gnd	Pin for connecting to GND on Arduino

The following figure shows the complete wiring:

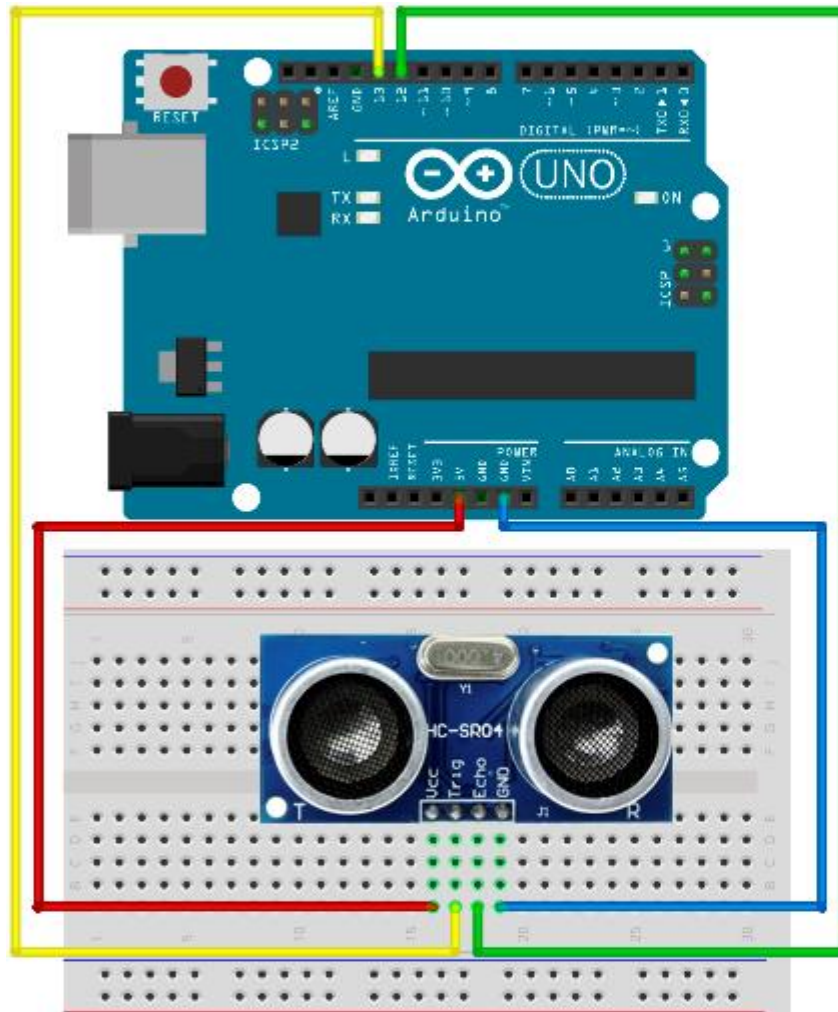


Figure 48: Ultrasonic distance sensor wiring

We will use the sensor to measure how far the nearest hard surface is and then display the distance to the serial port every second. In later examples, we will perform actions upon detecting objects at a certain distance but, for now, we will just look at what's going on. Do note that the sound doesn't bounce back very well from clothes, so the sensor might not read everything as you would expect. Here's the code for the example:

```
// we'll use trigPin for emitting
int trigPin = 13;

// we'll listen for the pulse on the echoPin
int echoPin = 12;

// variables for calculations
long duration, distance_inch, distance_cm;

void setup() {
  // initialize serial communication
  Serial.begin (9600);

  // initialize pins
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}

void loop() {
  // emit a pulse
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  // get the time that it takes for the pulse to return
  // pulseIn waits for the pulse and returns number of microseconds
  // divided by two because sound travels there and back again
  duration = pulseIn(echoPin, HIGH) / 2;

  // convert to inches and centimeters by using this formula
  distance_inch = duration / 74;
  distance_cm = duration / 29;

  // print the results
  Serial.print(distance_inch);
  Serial.print(" inch; ");
  Serial.print(distance_cm);
  Serial.println(" cm");
  Serial.println("");

  delay(1000);
}
```

If you open up the `Serial Monitor` tool, you should see how the distance changes as you put something in front of the distance sensor. You can even take a ruler and check the precision of the sensor. Do note that there might be minor differences depending upon what altitude you live and the weather conditions during your experiments. Also, the sensor can't detect when the objects are too close to it. It's also not possible to measure distances that are below 2cm (0.8 inch) and beyond 400 cm (157.5 inch). In the next section, we will perform actions if the object comes too near to the sensor.

Reacting On Approaching Objects

Most of the time, the sensors are not used just to measure how far something is. Usually we want some kind of action performed, such as turning on a light, starting a beep, or turning on an alarm. In this section, we will work on the previous example and use LEDs to mimic a simple alarm.

If everything is fine, the green LED will be on. But if something approaches too near to the sensor, we will turn the red LED on. The parts list will be a little longer than in the previous section.

Parts list for this section:

- Arduino Uno
- USB cable
- HC-SR04 Ultrasonic distance sensor
- 2x 100 Ohms Resistor
- 1x Red LED
- 1x Green LED
- Breadboard
- 8x Breadboard jumper wire

The wiring for the example is shown here:

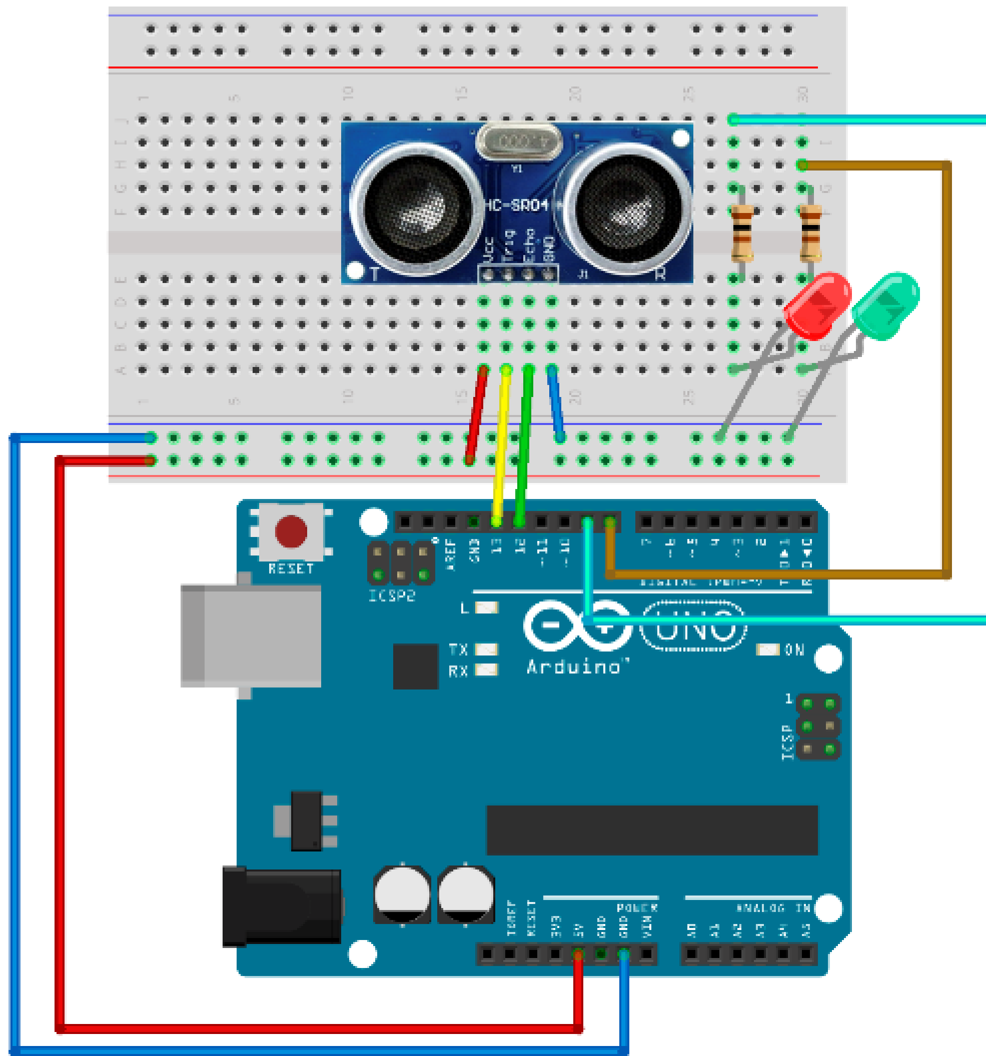


Figure 49: Distance alarm wiring

The code for the example:

```
// we'll use trigPin for emitting
int trigPin = 13;

// we'll listen for the pulse on the echoPin
int echoPin = 12;

int okPin = 8;
int nOkPin = 9;

// variables for calculations
long duration, distance_inch, distance_cm;

void setup() {
  // initialize pins
```

```
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(okPin, OUTPUT);
  pinMode(nOkPin, OUTPUT);
}

void loop() {
  // emit a pulse
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  // get the time that it takes for the pulse to return
  // pulseIn waits for the pulse and returns number of microseconds
  // divided by two because sound travels there and back again
  duration = pulseIn(echoPin, HIGH) / 2;

  // convert to inches and centimeters by using this formula
  distance_inch = duration / 74;
  distance_cm = duration / 29;

  // the distance is the same in both units
  // leave the one that suits you
  if (distance_cm < 28 || distance_inch < 11) {
    digitalWrite(nOkPin, HIGH);
    digitalWrite(okPin, LOW);
  }
  else {
    digitalWrite(okPin, HIGH);
    digitalWrite(nOkPin, LOW);
  }

  delay(200);
}
```

Tuning the Distance Sensor on the Fly

The previous example is always fixed to turn the warning LED on if the object comes closer than a predefined value. In most real-life situations, having a fixed value is not enough. There are situations in which we will want to tune the distance that our sensor is detecting. To overcome this shortcoming of our current solution, we are going to add a potentiometer to the game. We had a section a little while ago about how it works, so feel free to look back on it to refresh yourself with it a bit. We'll start with the parts list.

Parts list for this section:

- Arduino Uno
- USB cable
- HC-SR04 Ultrasonic distance sensor
- 2x 100 Ohms Resistor

- 1x Red LED
- 1x Green LED
- 10K Ohm potentiometer
- Breadboard
- 11x Breadboard jumper wire

We will regulate the alarm distance with the potentiometer. The minimum and the maximum values for the sensor are provided in the specification. We will use that in the example. We will also map the values that we read from the potentiometer to percentages. Then we will use the percentages to calculate the actual distance where the alarm is going to go off. We didn't mention it earlier, but the sensor will not be correct in all of the readings. If the reading is smaller or greater than what the sensor is capable of detecting, then we will simply skip a cycle for reacting and continue with the new one. The wiring is a bit more complicated than with the previous example:

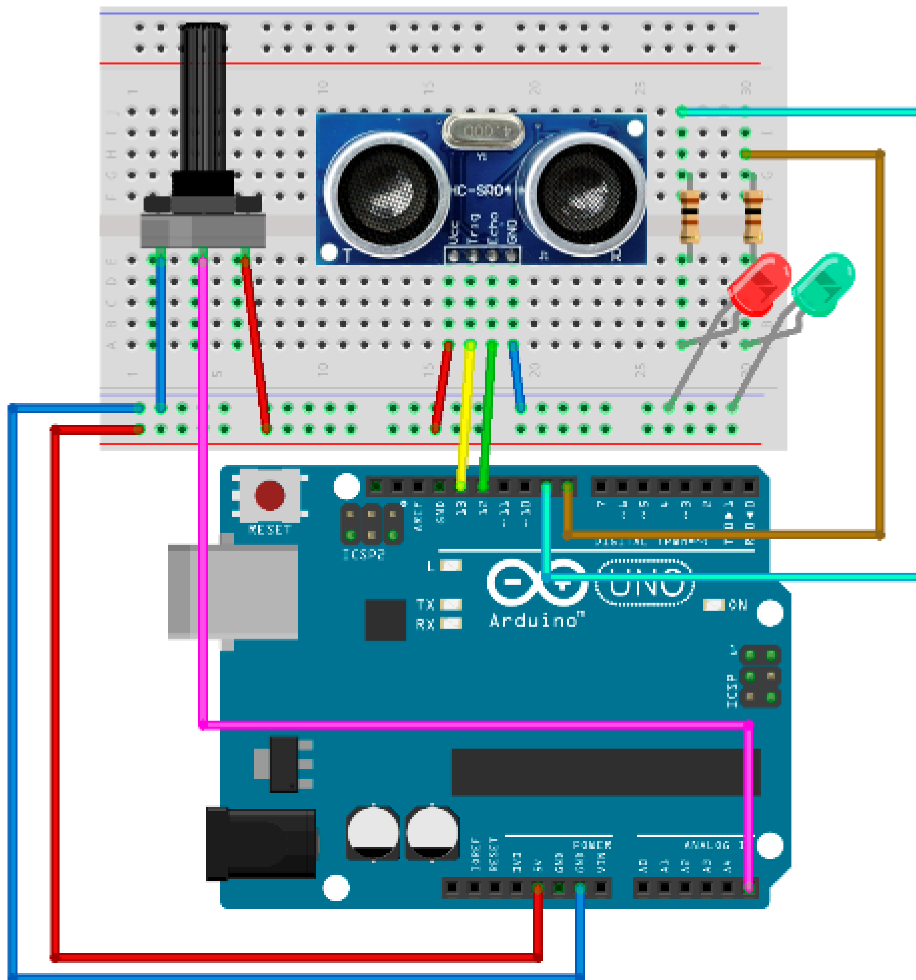


Figure 50: Distance alarm with regulating potentiometer

The code uses a bit more Arduino pins than in the previous sections. If you are using some other sensor for detecting the distance, please change the ranges in the code. This is the code listing for the example:

```
// we'll use trigPin for emitting
int trigPin = 13;

// we'll listen for the pulse on the echoPin
int echoPin = 12;

int okPin = 8;
int nOkPin = 9;

// sensor limits are defined in cm
int sens_min = 2;
int sens_max = 400;

// variables
long duration, distance, tune_in, alarm_distance;

void setup() {
  // initialize pins
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(okPin, OUTPUT);
  pinMode(nOkPin, OUTPUT);
}

void loop() {
  // check the current tuned setting
  tune_in = map(analogRead(A5), 1023, 0, 0, 100);

  alarm_distance = sens_min + ((sens_max - sens_min) * tune_in)/100;

  // emit a pulse
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  // get the time that it takes for the pulse to return
  // pulseIn waits for the pulse and returns number of microseconds
  // divided by two because sound travels there and back again
  duration = pulseIn(echoPin, HIGH) / 2;

  distance = duration / 29;

  // sanity check on the values
  if (distance > sens_max ||  distance < sens_min) {
    return;
  }

  // check if the alarm LED should light up
```

```
  if (distance <= alarm_distance) {
    digitalWrite(nOkPin, HIGH);
    digitalWrite(okPin, LOW);
  }
  else {
    digitalWrite(okPin, HIGH);
    digitalWrite(nOkPin, LOW);
  }

  delay(200);
}
```

When you are done with the wiring, change the potentiometer and see how the alarm starts to react to proximity. With this example, we covered the basic usages of the distance sensor. We have covered a lot but the examples were more or less stationary. In the next section, we will take a small step towards dynamic sensing.

Parking Sensor

We mentioned a parking sensor as one of the possible ways to use an ultrasonic distance sensor. In this section, we will go a step further and go through the steps needed to make one.

The wiring is a bit easier than in the previous example but the programming will be slightly complicated. The hardest part in this section will be the calculation of the beep intervals and making the buzzer beep the closer the object comes. As in the previous sections, we will start with the parts list.

Parts list for this section:

- Arduino Uno
- USB cable
- HC-SR04 Ultrasonic distance sensor
- 1x 100 Ohms Resistor
- Arduino compatible passive buzzer
- Breadboard
- 8x Breadboard jumper wire

The wiring for the example:

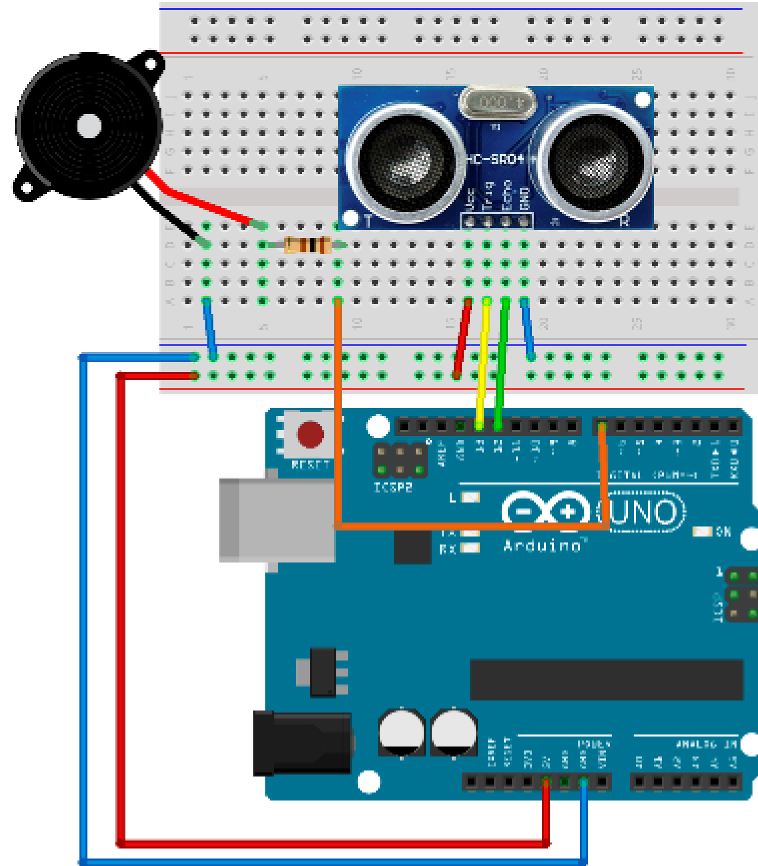


Figure 51: Park sensor wiring

The code is a bit more complicated than in the previous sections. The most complicated part is determining the interval between the beeps depending on the object distance:

```
// we'll use trigPin for emitting
int trigPin = 13;

// we'll listen for the pulse on the echoPin
int echoPin = 12;

// pin for the buzzer
int buzzPin = 7;

// sensor limitations defined in cm
int sens_min = 2;
int sens_max = 400;

// buzzer limitations are defined in cm
// it starts buzzing when something is closer than 100 cm
long buz_min = 2;
long buz_max = 100;

// beeping intervals in seconds
```

```
// this will be a long beep
int beep_max = 700;
// short beep
int beep_min = 50;

// variables for calculations
long duration, distance, beepInterval;
float pulsePercent;

// we'll initialize the lastBuzzTime to 0
long lastBuzzTime = 0;

void setup() {
  // initialize pins
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(buzzPin, OUTPUT);
}

void loop() {
  // emit a pulse
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  // get the time that it takes for the pulse to return
  // pulseIn waits for the pulse and returns number of microseconds
  // divided by two because sound travels there and back again
  duration = pulseIn(echoPin, HIGH) / 2;

  distance = duration / 29;

  // sanity check on the values
  if (distance > sens_max || sens_min < 2) {
    return;
  }

  // if distance is smaller than the beeping trigger distance, start beeps
  if (distance < buz_max) {
    // the easiest is to calculate the percentages of min and max distance
    pulsePercent = (100 * (distance - buz_min) / (buz_max - buz_min));

    // determine the beeping interval with the help of percentages
    beepInterval = beep_min + (pulsePercent * (beep_max - beep_min)) / 100;

    // we will emit a tone without using the delay function
    // so we will check the Arduino internal clock to check when to beep
    if ((millis() - lastBuzzTime) >= beepInterval) {
      tone(buzzPin, 2200, 100);
      // we have to remember the last time we started a beep so that we don't
      // start to emit a continuous sound
      lastBuzzTime = millis();
    }
  }
```

```
  // minimum sensing time is the minimum beep interval
  delay(beep_min);
}
```

Now try it out with the palm of your hand. The closer you get to the sensor, the time between the beeps get shorter. Note that the ultrasonic distance sensor works best on hard surfaces and that the beep might be a bit irregular if you are far away from the sensor. In that case, take any kind of object with a hard surface and try to see how the sensor reacts to it.

With the hard surface object, you should get much better results than with the palm of your hand. Also, the distance sensor sometimes doesn't detect surfaces that are parallel to it or at a greater angle because the sound beam bounces back in such a direction that it doesn't come back to the sensor. Most of the cars have multiple sensors in the fender because of that.

Also, some of the older sensors didn't detect small vertical obstacles such as poles and similar objects. But this sensor is a great fit for inexpensive projects and is often used in robotics. There are many Arduino robot designs out there that use this sensor to detect obstacles such as walls. The sensor is often mounted on turrets to give the robot more possibilities to detect obstacles. The sensor also has a very interesting shape because the emitter and the receiver look like eyes and the robot gets a very interesting form. In a way, the sensor represents its eyes.

Using Infrared Motion Sensor

Up until now, we have used an ultrasonic distance sensor in our examples. If you built the examples, you might have noticed that the ultrasonic distance sensor can't always detect objects that don't have a hard surface, or if the surface reflects the waves in such a way that they don't return back to the sensor. It also has relatively moderate capabilities in the sense of distance and can't detect something that's beyond four meters or 157 inches.

Also, sometimes it's not important how far something is, and we generally just want to detect whether or not somebody is in the room or if something is moving. An infrared sensor is a great fit for those situations, so we will use it in this section. We will build a movement detector with two LEDs. If nothing is moving, the green LED will be on. If something starts to move, we will turn the red LED on.

Parts list for this section:

- Arduino Uno
- USB cable
- Arduino-compatible infrared sensor
- 2x 100 Ohms Resistor
- 1x Green LED
- 1x Red LED
- Breadboard
- 7x Breadboard jumper wire

Wiring:

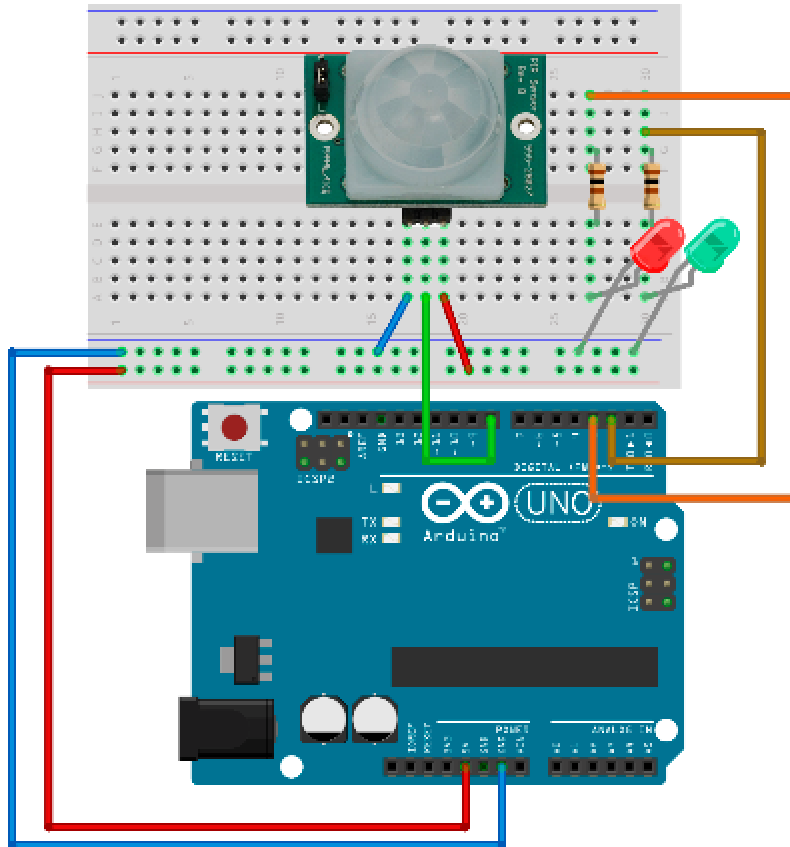


Figure 52: Motion sensor wiring

The sensor wiring may vary from producer to producer. Just to make sure, double check the wiring with the following table:

Table 4: Infrared Sensor Wiring for the Example

Pin	Connection
VCC	Pin for connecting 5V +
OUT	Connect to pin 8 in our example
GND	Pin for connecting to GND on Arduino

The code for the example is relatively simple. We will read the state from the input pin and then turn the LEDs on and off, depending upon what we read from the input:

```
// detection is going to be on pin 8
int detectPin = 8;

// LED pins
int greenPin = 2;
int redPin = 3;

int detectPinValue = LOW;

void setup() {
  // initialize pins
  pinMode(detectPin, INPUT);
  pinMode(greenPin, OUTPUT);
  pinMode(redPin, OUTPUT);
}

void loop() {
  // check the value from the sensor
  detectPinValue = digitalRead(detectPin);

  if (detectPinValue == HIGH) {
    // if sensor detected movement, turn the red LED on
    digitalWrite(greenPin, LOW);
    digitalWrite(redPin, HIGH);

    // we'll leave the red LED on for a while
    delay(500);
  }
  else {
    // turn the green LED on
    digitalWrite(greenPin, HIGH);
    digitalWrite(redPin, LOW);
  }

  // relatively short loop cycle
  delay(100);
}
```

This is a very interesting example. Move around the sensor to see how it reacts. Try not to move with your whole body but just move your hand to see what happens. This example is a nice basis for creating an alarm device. Another usage for this sensor is turning the light on so that we can see the path to the door without having to use a remote or something similar. In the next section, we will set up a motion sensor that turns an LED on when movement is detected, but only if the room is dark.

Turning the Light on Conditionally

One of the common usages of the motion sensor is to turn the light on and off when somebody approaches. The sensor works in the nighttime as well as in the daytime. But during the day, we would like to save electricity and not turn the light on because there is enough light. In this section, we will go through the wiring and the programming to do so. As with all of the examples, we will start with the parts list.

Parts list for this section:

- Arduino Uno
- USB cable
- Arduino-compatible infrared sensor
- 1x 10k Ohm resistor
- 1x GL5528 Photo Resistor or any 10K Ohm Photo Resistor
- 1x 100 Ohms Resistor
- 1x Yellow LED
- Breadboard
- 9x Breadboard jumper wire

The 10k Ohm resistor is paired with the photo resistor. The 100-Ohm resistor is paired with the LED as in the previous examples. We will use a yellow LED because it mimics the light bulb that's supposed to be turned on. We mentioned earlier that we would not show you any circuits with the strong currents, so here we are just using the LED. But the LED could easily be replaced with the relay and you would have the real thing. But at this stage of development, it's better not to play around with the dangerous currents. Let's have a look at the wiring:

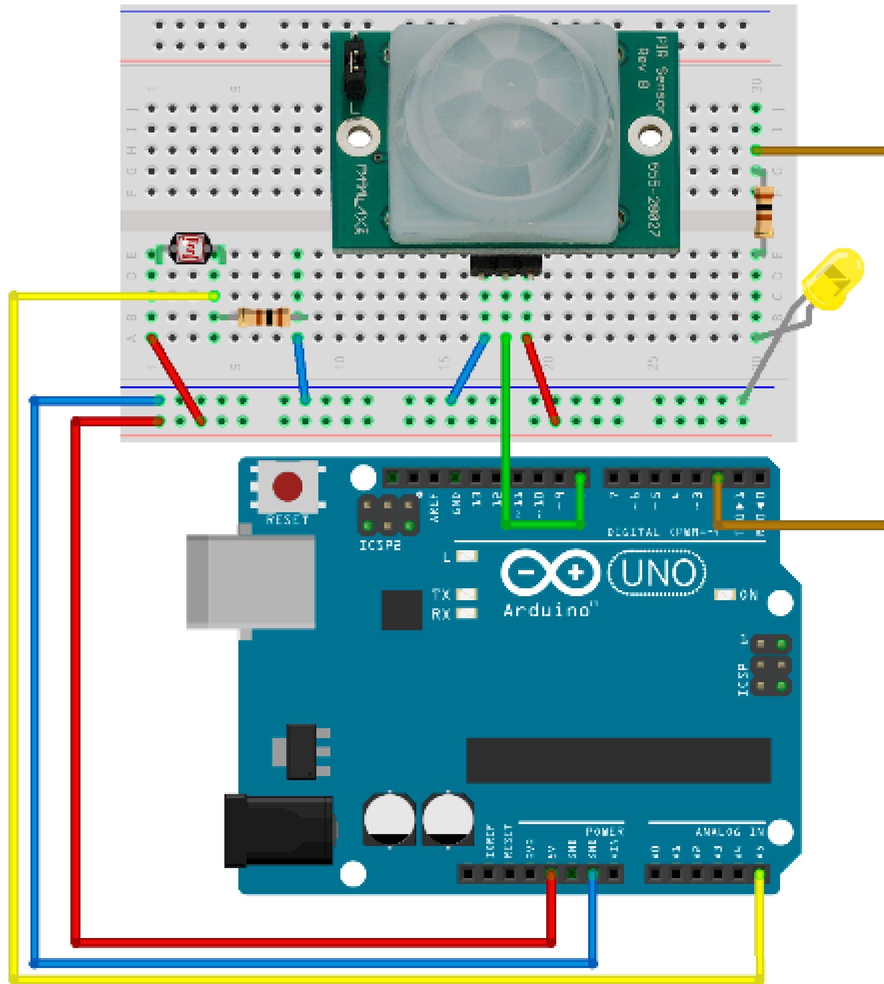


Figure 53: Conditional light on with motion sensor wiring

All that is missing is the code. We will make the timings a little bit shorter than in the real circuit. For instance, the light will remain on for around five seconds after the last registered movement. If you were building a real conditional light, we would leave the light on for couple of minutes after the last detected motion. Also, the software we are making here would definitely be in a higher price range because the sensor monitors for the last movement. Usually the lighting solutions just turn themselves on and wait no matter what, and after the time expires, they turn themselves off and wait for the new movement again. This example constantly monitors the space and continues to provide the light five seconds after the last movement is registered. If you are trying the example in a very light or dark room, tune the `startTurningLightAt` variable to your needs. It's best that you experiment with the value and see what happens:

```
// movement detection is going to be on pin 8
int detectPin = 8;
int detectPinValue = LOW;

// light pin is going to be on pin2
int lightPin = 2;
```

```
// we'll store reading value here
int lightLevel, lightReading;

// give this a bigger value in a light room
int startTurningLightAt = 70;

long lastMovementTime = 0;

// continue with light 5 seconds after last movement
int continueLightTime = 5000;

void setup() {
  pinMode(lightPin, OUTPUT);
}

void loop() {
  // read the light level from pin A5
  lightReading = analogRead(A5);

  // we'll measure the input values and map 0-1023 input
  // to 0 - 100 values
  lightLevel = map(lightReading, 0, 1023, 0, 100);

  if (lightLevel < startTurningLightAt) {
      detectPinValue = digitalRead(detectPin);

      if (detectPinValue == HIGH) {
        digitalWrite(lightPin, HIGH);
        lastMovementTime = millis();
      }
      else if ((millis() - lastMovementTime) > continueLightTime) {
        digitalWrite(lightPin, LOW);
      }
  }
  else {
    digitalWrite(lightPin, LOW);
  }

  delay(100);
}
```

This is the last example in the detecting objects section. For the better part of the chapter, we have been using two of the most popular sensors in the Arduino world: the ultrasonic distance sensor and the infrared movement sensor. We also showed the classical examples seen in everyday life involving those two. In the next chapter, we will give an overview of the most common Arduino networking solutions.

Chapter 7 Networking

All of the examples up until now were offline. We didn't send the readings from the Arduino to another device nor did we use the Arduino to receive information over the networks to start to behave differently. There are two types of connections that the Arduino usually employs: point-to-point and network-aware. Point-to-point usually has shorter range and is used very often in home automation. We use network-aware connections if we want to receive something from or send something to a greater distance. Some of the examples in this section will require you to have two Arduino boards. If you don't have a second Arduino board, you can just follow along and see what's possible if you get the second Arduino.

Communication with MK Modules

One of the simplest means of communication in the Arduino world is the point-to-point communication with the MK modules. The modules are always in pair and the communication goes in one way only. The range on the modules is relatively reasonable for communication between devices located in rooms around a household, but only if you solder on an antenna. The antenna can be an ordinary insulated wire. The tricky part is to calculate the length of the wire. The best results are achieved when the antenna is quarter of the wavelength. The wavelength is determined by the frequency at which the transmission is done. The wavelength can be calculated by a formula.

Note: wavelength = speed_of_light / frequency

The MK modules usually work at 315, 330, and 433 MHz, so perhaps it would be best to put it in tabular form:

Table 5: Antenna Lengths by Frequency

Frequency	Antenna Length in Centimeters	Antenna Length in Inches
315 Mhz	23.8	9.4
330 Mhz	22.7	8.9
433 Mhz	17.3	6.8

You can use the previous table to determine the antenna length without going too deep into calculations. We won't need the antenna in our example because it would require soldering, and soldering is outside of the scope of this book. Instead, we will put the transmitter and the receiver near each other. It's very important that you differentiate which module is the transmitter and which module is the receiver. Each has a very different role as well as appearance:

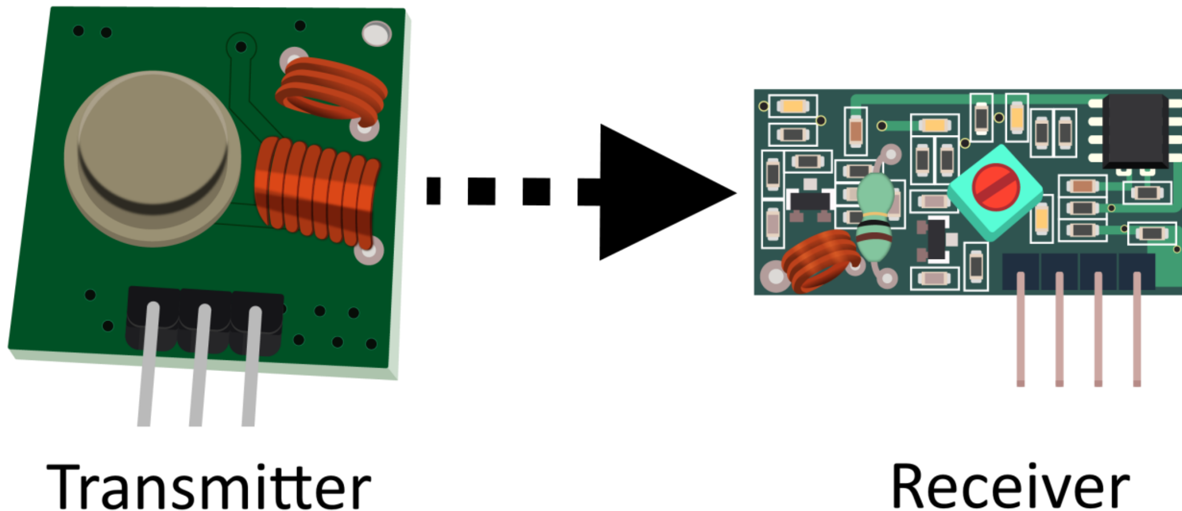


Figure 54: MK transmitter on the left, MK receiver on the right

The module pair does not support two-way communication. If you need two-way communication, you have to use two pairs with a transmitter and a receiver on both sides. The wiring is relatively simple, as we will use only three wires per module. The receiver has four pins, two of them are used for receiving the data. But using two pins for receiving is not reliable because of interference between them. It's important that the frequencies of the transmitter and receiver match. It doesn't matter if it's 315, 330, or 433 MHz; it's just important that they are the same. Let's have a look at the parts list for this section.

Parts list for this section:

- 2x Arduino Uno
- 2x USB cable
- XY-MK-5V Receiver module
- FS1000A Transmitter module
- 6x Breadboard Male-to-female jumper wire

We will use one Arduino to send a message to the communications channel. Then, we will use the second Arduino to pick up the message from the channel and send it to the serial port on the computer. Communicating and implementing protocols is usually done with libraries. It is possible to send relatively simple signals with the transmitter and the receiver module without using libraries, but handling complicated messages and checking for errors is a chapter of its own. One of the best libraries for the MK transmitter and the receiver is called **`VirtualWire`**. To start using this, you have to download the archive. The archive is located here. Download it and remember where you saved it.

Create a new sketch and click `Sketch` > `Include Library` > `Add Zip Library`. Select the zip archive that you downloaded and the IDE should automatically install the library. If you already installed the library, there is no need for you to install it twice. The Arduino IDE will probably report an error if you try to do so. There is no breadboard in this example, so we'll show how to wire the modules with tables instead. There are visible markings on the module so we will just go over the connections necessary for our example to work. The transmitter module works on a smaller voltage of 3.3 volts. There are modules available that can receive up to 12 volts, but please check the specification of the module that you purchased just to be on the safe side. Connecting the equipment with voltages that are too big can damage it. There are only three wires on the transmitter module:

Table 6: Transmitter Wiring

Transmitter PIN	Arduino Uno PIN
ATAD (reverse for DATA)	12
VCC	3.3 V
GND	GND

The receiver module has four pins, but the best results are when we use only three wires. Let's have a look at the pins:

Table 7: Receiver wiring

Receiver PIN	Arduino Uno PIN
VCC	5 V
DATA (next to VCC)	11 – Connect just one data pin!
DATA (next to GND)	11 – Connect just one data pin!
GND	GND

When connecting the receiver, connect just one of the data pins to Arduino pin 11. Now that the wiring is set up, we will interchange messages and then print the messages to the Serial port. We will have a look at the transmitter code first:

```
#include <VirtualWire.h>
// connect the ATAD (data reverse) to pin 12

// greater speed smaller reliability
// smaller speed greater reliability
// 2000 bits per second is fine for most of the applications

int bits_per_second_speed = 2000;

void setup() {
  // initialize VirtualWire
  vw_setup(bits_per_second_speed);
}

void loop() {
  send("This Totally Works!");
  delay(1000);
}

void send (char *message) {
  // send the message
  vw_send((uint8_t *)message, strlen(message));

  // wait until the message is transmitted
  vw_wait_tx();
}
```

The receiver code:

```
#include <VirtualWire.h>
// connect just one of the DATA wires to pin 11

// transmission speeds have to match
int bits_per_second_speed = 2000;

// buffer for storing incoming messages
byte message[VW_MAX_MESSAGE_LEN];

// we'll save received message size, initial value is max
byte messageLength = VW_MAX_MESSAGE_LEN;

void setup() {
  // initialize serial communication with the computer
  Serial.begin(9600);

  // just to let us know that something is going on
  Serial.println("Initializing device");
  // initialize
  vw_setup(bits_per_second_speed);
  // starting the receiver
  vw_rx_start();
```

```
  Serial.println("VirtualWire receiver started ...");
}

void loop() {
  // not blocking
  if (vw_get_message(message, &messageLength)) {
    Serial.print("Incoming: ");

    // print every byte
    for (int i = 0; i < messageLength; i++) {
      Serial.write(message[i]);
    }

    Serial.println();
  }
}
```

This example might not look like something special, but, believe it or not, the previous example is a basis for a lot of Arduino-based DIY solutions, ranging from home-based automation to security and remote sensing.

The main advantage of the module is that it is extremely cheap and has a relative good range with a soldered-on antenna. The antenna can be something as simple as a wire with a defined length. There are even cases in which people claim that they managed to get the information up to 100 meters or 110 yards.

In reality, it is really dependent on the wall type that you have in between. If it's a thick, reinforced concrete wall, you probably won't go through it. But for a regular home and a reasonable distance, the MK modules are just fine. In the next section, we will look at a more robust piece of wireless technology that works on much higher speeds and can communicate both ways (and not always from transmitter to the receiver).

Using nRF24L01+ Data Transceivers

With the nRF24L01+, the data can go both ways at the same time. There are no dedicated emitter and receiver circuits. The frequency at which the transceivers operate is 2.4 GHz and is in the range of the usual home wireless Internet access equipment.

Compared to the MK modules from the previous section, the most difficult thing with them is the wiring. Wiring is also dependent upon the library that we are using. There are several libraries available. The most commonly used one at the time of writing is the **`RF24`**.

To install the library, download the file from Github here and remember where you saved it. Create a new sketch and click **`Sketch`** > **`Include Library`** > **`Add ZIP Library`**. Select the zip archive that you downloaded and the Arduino IDE should automatically install the library. If you decide on some other library, please check how the module should be connected. Next, we will go through the wiring scheme. Let's have a look at the module:

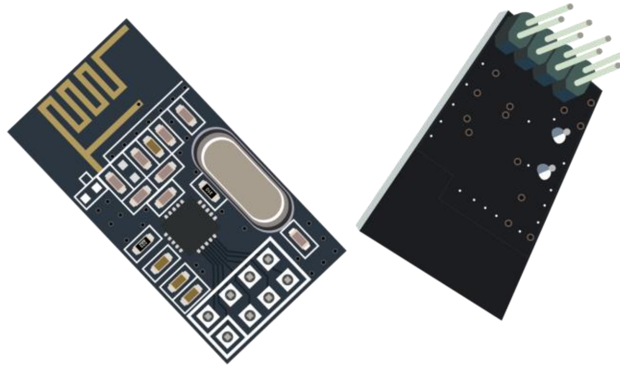


Figure 55: nRF24L01+ front and back side view

The easiest way to connect the pins is to turn the module so that you see the backside of it. The backside is the side with the pins sticking out:

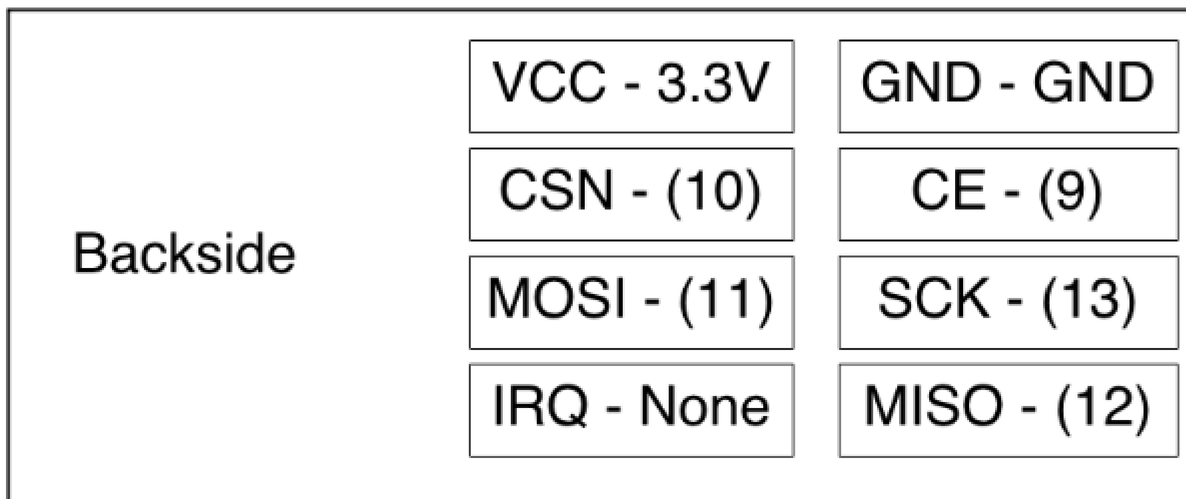


Figure 56: nRF24L01+ wiring

There are many wiring schemas available online, but most of them require you to look for the pin from the other side. This image represents how you see the pins when you turn it to the backside. It will be a lot easier for you to connect the module this way. There are some versions of the nRF24L01+ available online with this layout printed on the backside next to the pins. Use the previous figure if you don't have the version with the print. Next to the name of the pin is the name of the destination pin on the Arduino. Here are detailed explanations with every pin on the transceiver module:

Table 8: nRF24L01+ Pin Description of Pins and Wirings with the Arduino

Receiver PIN	Description	Destination Pin on Arduino
VCC	Pin for 3.3V +	3.3 V
GND	Ground	GND
CSN	Chip Select Not. If this is low, chip responds to SPI commands. Serial Peripheral Interface (SPI) Bus is a serial synchronous communication standard.	10
CE	Chip enable. If it's high, the module is sending or listening.	9
MOSI	Master-Out-Slave-In. Sends the data from microcontroller to the device.	11
SCK	SPI Shift Clock; used for synchronization of data during the transfer.	13
IRQ	Optional Interrupt Request pin.	This pin is not connected in our examples.
MISO	Master-In-Slave-Out. Sends the data from the device to the microcontroller.	12

We will need two Arduino boards in this example and two transceiver modules. We will also need 14 wires to connect both Arduino boards. We won't make very complex protocols in the example. Every Arduino will read the data and send the data.

We will adapt one of the Arduino boards to display the information in the Serial Monitor so that we know what's going on. We will give names to the microcontrollers. The first one will get a simple designation "A". The data that we will send over will be the basis for a simple protocol. We will put the source and the destination info into packages plus some randomly generated data like temperature and light. We won't set up the wiring to measure the values because the focus in this example is on the networking part.

```
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

// set up nRF24L01 radio on SPI bus plus pins 9 & 10
RF24 radio(9, 10);

// channel designations, put as many as you need, 2 are enough for this
// every transceiver can have one out and up to 5 read channels
const uint64_t pipes[2] = { 0xF0F0F0F0E1LL, 0xF0F0F0F0D2LL };

typedef struct {
  char source;
  char destination;
  float temperature;
  int light;
} TotallyCustomData;

TotallyCustomData data_out, data_in;

void setup(void) {
  Serial.begin(9600);
  radio.begin();
  // 15 millis delay for channel to settle, try to send 15x
  radio.setRetries(15, 15);

  radio.openWritingPipe(pipes[0]);
  radio.openReadingPipe(1, pipes[1]);

  radio.startListening();
}

void loop(void) {
  data_out.source = 'A';
  data_out.destination = 'B';
  data_out.temperature = random(30);
  data_out.light = random(100);

  // check if the data is available
  if (radio.available()) {
    bool done = false;

    // read the data until finished
    while (!done) {
      done = radio.read(&data_in, sizeof(data_in));

      // print the data
      Serial.println("Received data");
      Serial.print("source = ");
      Serial.println(data_in.source);
      Serial.print("destination = ");
      Serial.println(data_in.destination);
      Serial.print("temperature = ");
```

```
      Serial.println(data_in.temperature);
      Serial.print("light = ");
      Serial.println(data_in.light);
      Serial.println("");
    }
  }

  // send the data after reading is done

  // stop listening so we can talk.
  radio.stopListening();
  bool ok = radio.write(&data_out, sizeof(data_out));
  radio.startListening();

  // just so that we don't send too much data
  delay(1000);
}
```

Let’s look at the code that we are going to put on the Arduino board “B”:

```
#include <SPI.h>
#include "nRF24L01.h"
#include "RF24.h"

// set up nRF24L01 radio on SPI bus plus pins 9 & 10
RF24 radio(9, 10);

// radio pipe addresses for the two nodes to communicate.
const uint64_t pipes[2] = { 0xF0F0F0F0E1LL, 0xF0F0F0F0D2LL };

typedef struct {
  char source;
  char destination;
  float temperature;
  int light;
} TotallyCustomData;

TotallyCustomData data_out, data_in;

void setup(void) {
  radio.begin();
  // 15 millis delay for channel to settle, try to send 15x
  radio.setRetries(15, 15);

  radio.openWritingPipe(pipes[1]);
  radio.openReadingPipe(1, pipes[0]);

  radio.startListening();
}

void loop(void) {
  data_out.source = 'B';
```

```
  data_out.destination = 'A';
  data_out.temperature = random(30);
  data_out.light = random(100);

  if (radio.available()) {
    bool done = false;
    while (!done) {
      done = radio.read(&data_in, sizeof(data_in));
    }
  }

  // after data reading, write something
  // transceiver works in both directions, so cool :)

  // stop listening so we can talk.
  radio.stopListening();
  bool ok = radio.write(&data_out, sizeof(data_out));
  radio.startListening();

  // just so that we don't send too much data
  delay(1000);
}
```

Be careful when uploading the examples because it can easily happen that you upload the code to the wrong board! Double-check the Serial ports before uploading the sketch to the Arduino board.

The transmission distance depends upon the wall setup that you have at your home. So there will be border situations where you will have a feeling that you are missing just a few more inches for everything to work. Try to reduce the length of the package in those situations. In our example, the message contains eight bytes but the payload size is by default set to 32 bytes. Try reducing the payload length to eight in the **`setup`** function.

```
void setup(void) {
…
    radio.setPayloadSize(8);
…
```

One other problem that appears on the Arduino Uno boards is that the voltage on the 3.3V pin is not very stable and, as soon as the transmission starts, there is a very high message loss rate. Some people are using a separate 3.3V power source.

Some solder a 10-μF capacitor across 3.3V and the GND pin on the transceiver. Message loss rate is reduced by 25 percent with the capacitor. There are also versions of the module that have a larger antenna similar to one available on home routers. The module used in the example has an antenna printed on the board of the transceiver.

Connecting to Wireless with ESP8266 Chip

There is a lot of talk and articles about the Internet of Things (IoT), the devices connecting to the IoT, and the exchanging of data with other devices and computers. The MK chips and the nRF24 chips mentioned in the previous sections usually support an infrastructure in which we have one home hub that gathers the signals from the other devices and then uses the messages coming in from those device to send them further to other devices, with more computing power inside the home network or somewhere online.

Most Internet access today is over the wireless router available in our homes. Now, the question arises, wouldn't it be great if we could make our smart devices use the already available wireless infrastructure to send the data around? Up until recently, those devices were pretty expensive and it kind of defeated the whole purpose if those devices were more expensive than the Arduino board. Actually, a new member was recently added to the Arduino family that can do all that, namely, the Arduino Yún. But the price mark for it is more than two times greater than the Arduino Uno. An excellent choice with a very reasonable price is the ESP8266 Wi-Fi transceiver. It actually looks very similar to the nRF24 chip:

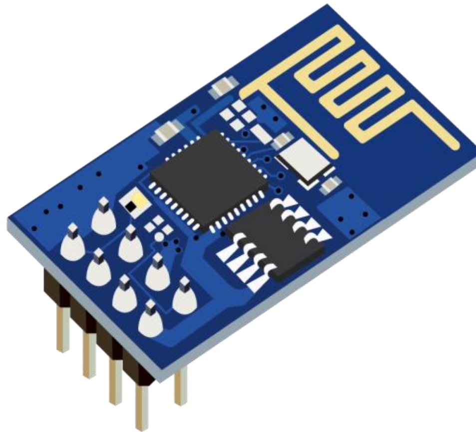


Figure 57: ESP8266 front side view

In this example, we will use multiple wires, the proto-board, the ESP8266 module, and the Arduino Uno. The ESP8266 is a 3.3 V circuit, and you should always connect all of the pins to 3.3 V sources as none of the pins is 5 V compatible. All of the pins from now on are stepped down to 3.3 V when connected, even if the text states connect to Arduino pin.

Parts list for this section:

- 1x Arduino Uno
- 1x USB cable
- ESP8266 wireless chip
- Breadboard
- 2x 1K-Ohm resistor
- 2x 3.3V Zener diode, for stepping down the voltage from Arduino 5V to chip 3.3V
- 5x Breadboard male-to-female jumper wire

- 6x Breadboard male-to-male jumper wire
- A working home wireless network

The pin names on the EXP8266 are shown on the following figure:

Backside		
	GND - GND	TX - (3)
	GPIO 2	CH_PD - 3.3V
	GPIO 0	RESET
	RX - (2)	VCC - 3.3 V

Figure 58: ESP8266 backside side view with pin names

Let's jump to wiring right away:

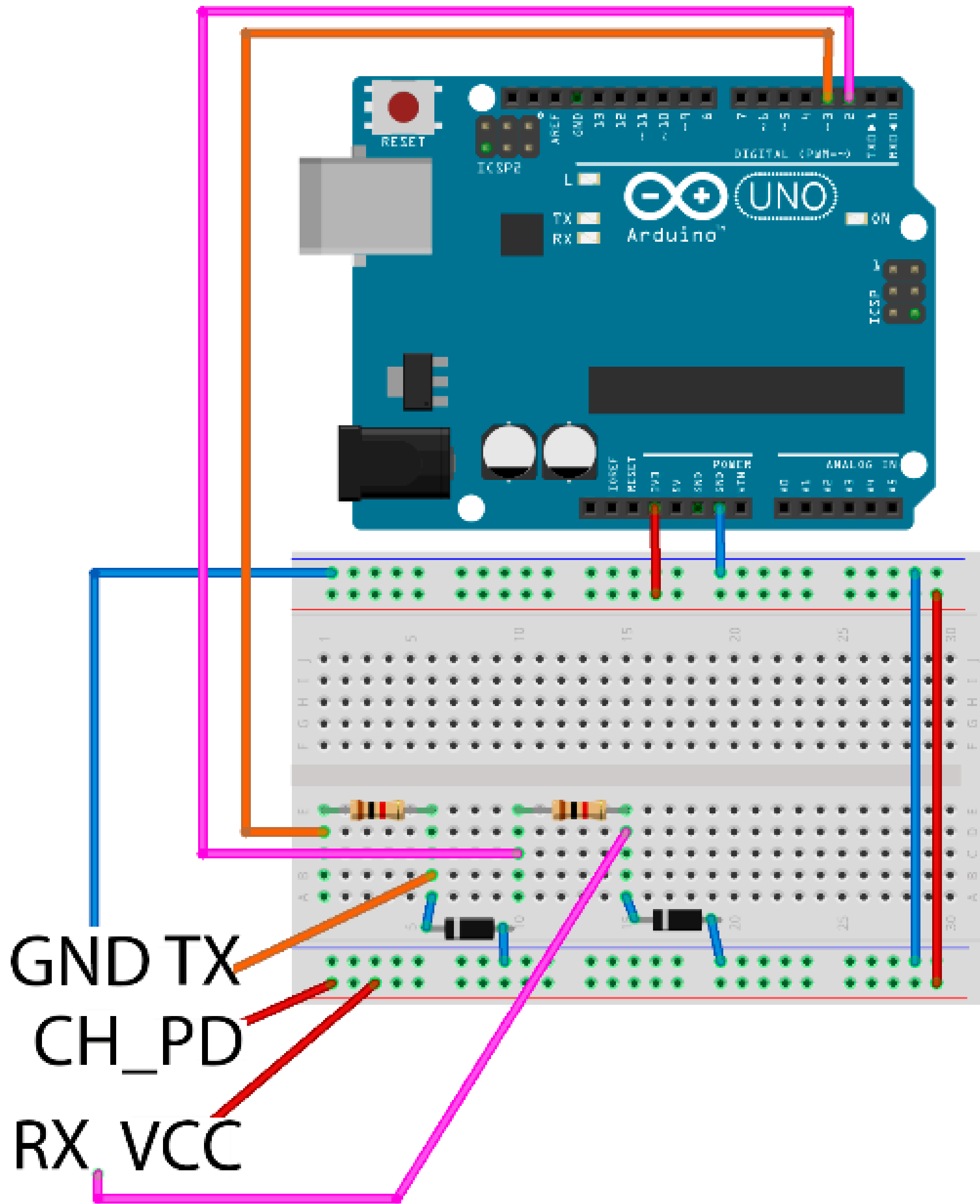


Figure 59: ESP8266 wiring

Unlike the previously used chip, this chip requires only five wires to get connected. The pins that need to get connected are marked with a blue color in the previous figure. The chip works so that we send it textual commands over the serial port, but the command list is very long, and it's a bit tedious work, so it's best that we use a library. The library is available for download on Github [here](). Create a new sketch and click **`Sketch`** > **`Include Library`** > **`Add ZIP Library`**. Select the zip archive that you downloaded and the Arduino IDE should automatically install the library.

However, the library will not work out of the box and there is a little bit of tweaking that we need to do before it is suitable for our use. But, before doing that, let's take a short step back. The Arduino uses TX and RX pins for serial communication. We use those two pins to transfer the programs to the board and to receive messages from the Arduino. So, if we want the Arduino to communicate over serial with other devices, we have to use an Arduino library called **`SoftwareSerial`** that allows us to turn any pair of pins into serial communication pins.

In our example, we will use the real Arduino TX and RX pins to show the messages in the Serial Monitor, and we will turn pin 3 into the software version of RX and pin 2 into the software version of TX. The software serial library comes with the Arduino so you don't have to install additional libraries. There is a little tweak that we have to make in the **`WeeESP8266`** library before we use it in our examples. First, you have to find the folder where the Arduino libraries are located:

Table 9: Arduino libraries folder location by platform

Platform	Location
Windows	My Documents\Arduino\libraries\
Linux	Documents/Arduino/libraries/
Mac	Documents/Arduino/libraries/

Then, you have to find the folder inside the libraries folder where the newly installed library is located. It is called **`ITEADLIB_Arduino_WeeESP8266-master`** or **`ITEADLIB_Arduino_WeeESP8266`**. Open the file called **`ESP8266.h`** with a raw text editor. Look for the following line:

```
// #define ESP8266_USE_SOFTWARE_SERIAL

turn this into:

#define ESP8266_USE_SOFTWARE_SERIAL
```

Remove the comments from the beginning of the line. If this line remains commented, you will not be able to upload the sketch to the Arduino board because the following example will not compile. We will start with something relatively simple since, for starters, we just want our Arduino to access our wireless network and obtain an IP address:

```
#include <ESP8266.h>
#include <SoftwareSerial.h>

// define the access data for your home wifi
#define SSID        "YourSSID – Name of your wifi connection"
```

```
#define PASSWORD    "YourWirelessPassword"

// initialize a softwareserial with rx - 3 and tx - 2
SoftwareSerial mySerial =  SoftwareSerial(3, 2);

// initialize the wifi to work with mySerial
ESP8266 wifi(mySerial);

void setup(void) {
  Serial.begin(9600);
  Serial.println("Setup begin");

  Serial.print("FW Version: ");
  Serial.println(wifi.getVersion().c_str());


  if (wifi.setOprToStation()) {
    Serial.println("to station ok");
  } else {
    Serial.println("to station err");
  }

  if (wifi.joinAP(SSID, PASSWORD)) {
    Serial.println("Join AP success");
    Serial.print("IP: ");
    Serial.println(wifi.getLocalIP().c_str());
  } else {
    Serial.println("Join AP failure");
  }

  Serial.println("setup end");
}

void loop(void) {
  // we'll leave the loop empty
}
```

If everything went fine, you should see an IP address that your router has assigned you. This is sort of a simple Hello World for this component. But, if you take a step back and look at all of the possibilities with the protocols and emission standards, getting everything to connect to a home wireless network, this example is really something. Your Arduino board is becoming an actual IoT device, just like that.

Getting Data Online with ESP8266 Chip

The wiring and the components are the same as in the previous section. The focus will be more on the programming. The task is to send the data from the Arduino over a TCP to a remote server. To make it simple, we are just going to display the sent data on the server side. In this example, we will create a simple server, too. This is a bit out of scope for the book, and we didn't mention any Python programming up until now, but please try to find a computer with Python installed or install it by following the instructions available here. Note that, on some systems, you will have to add the python executable to the PATH variable, but that is all outside of the scope of this book. Please follow the official documentation available under the previous link. After that, create a file called **simple_service.py** and remember the location where you saved it on the disk:

```
#!/usr/bin/env python

"""
A simple server
"""

import socket

host = '0.0.0.0'
port = 31233
backlog = 10
size = 1024

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
s.listen(backlog)

# do until script is interrupted
while 1:
    # accept incoming connections
    client, address = s.accept()
    # get the sent data
    data = client.recv(size)
    # print received data
    print("request: " + data)
    # close the connection
    client.close()
```

If you have Python installed, you can just run the script from the command line:

```
$ python simple_service.py
```

This covers the server side of the example. Also, be very careful with the firewall setting. If there is a firewall between the server and the ESP chip, the example will not work. Now that the server side is taken care of, let's look at the Arduino code:

```
#include <ESP8266.h>
#include <SoftwareSerial.h>

// define the access data for your home wifi
#define SSID        "YourSSID"
#define PASSWORD    "YourPassword"

// initialize a softwareserial with rx - 3 and tx - 2
SoftwareSerial mySerial(3, 2);

// the endpoint of the service
// in my case it's 192.168.1.2
#define HOST_NAME   "192.168.1.2"
#define HOST_PORT   (31233)

ESP8266 wifi(mySerial);

void setup(void) {
    Serial.begin(9600);
    Serial.println("Setup begin");

    Serial.print("FW Version:");
    Serial.println(wifi.getVersion().c_str());

    if (wifi.setOprToStationSoftAP()) {
        Serial.println("to station + softap ok");
    } else {
        Serial.println("to station + softap err");
    }

    if (wifi.disableMUX()) {
        Serial.println("single ok");
    } else {
        Serial.println("single err");
    }

    Serial.println("setup end");
}

void loop(void) {
    uint8_t buffer[128] = {0};

    if (wifi.createTCP(HOST_NAME, HOST_PORT)) {
        Serial.println("create tcp ok");
    } else {
        Serial.println("create tcp err");
    }

    char *request = "This could be anything!";
    Serial.println(request);

    wifi.send((const uint8_t*)request, strlen(request));
```

```
    uint32_t len = wifi.recv(buffer, sizeof(buffer), 1000);

    if (len > 0) {
        Serial.print("Received:[");
        for(uint32_t i = 0; i < len; i++) {
            Serial.print((char)buffer[i]);
        }
        Serial.println("");
        Serial.println("]");
    }

    delay(10000);
}
```

If everything went fine, you should see the data coming in from the Arduino. In our example, it will look something like the following:

```
$ python echo_service.py
request: This could be anything!
```

This was an example in which the ESP chip is sending the data to a server. But ESP is actually a really powerful chip. It even has its own firmware and you can upgrade it if you want. There are two GPIO pins available on the board and there are a lot of examples online where the Arduino is not even needed and the chip can function on its own. To prove how powerful this chip actually is, we are going to build a small TCP echo server that reverses back the data sent to it. Here is the code for the Arduino:

```
#include <ESP8266.h>
#include <SoftwareSerial.h>

// define the access data for your home wifi
#define SSID        "YourSSID"
#define PASSWORD    "YourPassword"

// initialize a softwareserial with rx - 3 and tx - 2
SoftwareSerial mySerial(3, 2);

ESP8266 wifi(mySerial);

void setup(void) {
  Serial.begin(9600);
  Serial.println("setup begin");

  Serial.print("FW Version:");
  Serial.println(wifi.getVersion().c_str());

  if (wifi.setOprToStationSoftAP()) {
    Serial.println("to station + softap ok");
  } else {
    Serial.println("to station + softap err");
```

```
  }

  if (wifi.joinAP(SSID, PASSWORD)) {
    Serial.println("Join AP success");
    Serial.print("IP: ");
    Serial.println(wifi.getLocalIP().c_str());
  } else {
    Serial.println("Join AP failure");
  }

  if (wifi.enableMUX()) {
    Serial.println("multiple ok");
  } else {
    Serial.println("multiple err");
  }

  if (wifi.startTCPServer(8090)) {
    Serial.println("start tcp server ok");
  } else {
    Serial.println("start tcp server err");
  }

  if (wifi.setTCPServerTimeout(10)) {
    Serial.println("set tcp server timout 10 seconds");
  } else {
    Serial.println("set tcp server timout err");
  }

  Serial.print("setup end, ready to receive requests");
}

void loop(void) {
  uint8_t buffer[128] = {0};
  uint8_t buffer_reverse[128] = {0};
  uint8_t mux_id;
  uint32_t len = wifi.recv(&mux_id, buffer, sizeof(buffer), 100);
  if (len > 0) {
    Serial.print("Status:[");
    Serial.print(wifi.getIPStatus().c_str());
    Serial.println("]");

    Serial.print("Received from :");
    Serial.print(mux_id);
    Serial.print("[");
    for (uint32_t i = 0; i < len; i++) {
      Serial.print((char)buffer[i]);
    }
    Serial.println("]");

    for (int c = len - 1, d = 0; c >= 0; c--, d++) {
      buffer_reverse[d] = buffer[c];
    }

    if (wifi.send(mux_id, buffer_reverse, len)) {
```

```
      Serial.println("send back ok");
    } else {
      Serial.println("send back err");
    }

    if (wifi.releaseTCP(mux_id)) {
      Serial.print("release tcp ");
      Serial.print(mux_id);
      Serial.println(" ok");
    } else {
      Serial.print("release tcp");
      Serial.print(mux_id);
      Serial.println(" err");
    }

    Serial.print("Status:[");
    Serial.print(wifi.getIPStatus().c_str());
    Serial.println("]");
  }
}
```

When you start the example, you should see what IP address was assigned to the Arduino with the ESP chip in the **`Serial Monitor`** tool. Then, we will use the Telnet tool to connect to it on the port 8090. The ESP should return reversed characters that we type in. Remember, there is a 10-second timeout for you to type something in and hit Enter:

```
$ telnet 192.168.1.5 8090
Trying 192.168.1.5...
Connected to unknown-18-fe-34-9b-79-87.lan.
Escape character is '^]'.
!sklof lla s'tahT

That's all folks!Connection closed by foreign host.
```

This example is the final one for showing the capabilities of the ESP module and it also concludes the part about the Networking with the Arduino. This is also the last example in the book.

Chapter 8 Conclusion

Before the Arduino project was started in 2005, students in Italy had to pay $100 for a microcontroller called “BASIC Stamp.” The inventor of the Arduino wanted something that would fit into a student’s budget, so the main guideline was to build something similar but for no more than $30. Today, an Arduino Uno board costs around $25 and there are 10 or more commercially available variants of the Arduino Board, and probably millions of Arduino boards and their variants are in facilities all over the world.

We can find Arduino boards in homes and they are used for various kinds of projects from home automation to home security or even brewing a beer. Arduino was launched into space with project ArduSat, and students all over the world can make experiments with it. It is even used at CERN LHC in the equipment that was used to confirm the existence of the Higgs boson.

The possibilities to use the Arduino are literally infinite. There are billions of people around the world and a fair amount of them have probably, at some point in their lives, imagined a device not available on the market or a device that was too expensive that could help them in some aspect of their lives. I was always interested in electronics, but I didn’t have years to learn all there is about them just so that I could build stuff that was interesting to me.

At one point, there were a lot of posts on forums and articles on blogs so I thought perhaps it’s time to buy one. Not much had changed when the Arduino UNO board arrived. It collected dust for almost a year and that is a very long time by today’s standards. Actually, just looking at the board scared me; all I could think of is all those books on electronics with so much detailed information! Then, one evening, carefully following the online tutorials, I started to work with it. By the end of the evening, I was amazed at how easy it all was. In one single evening, I did more than with all of the information I gathered before in the literature. It is incredible how easy it is to start with the electronics if you use the Arduino Board, and how much is abstracted from you if you use it. In essence, those were the humble beginnings that led to my writing of this book.

If you followed along up until now, I hope you enjoyed the materials in the book and that you are going to use it to build things that you always wanted to build but didn’t have the tools before to do so. Arduino technology really makes dreams come true and the imagination come to life. Massimo Banzi, one of the Arduino inventors, once said that Arduino is open-sourcing imagination. Don’t stop dreaming your dreams, and remember, you now have a powerful tool in your toolbox to help you pursue them.