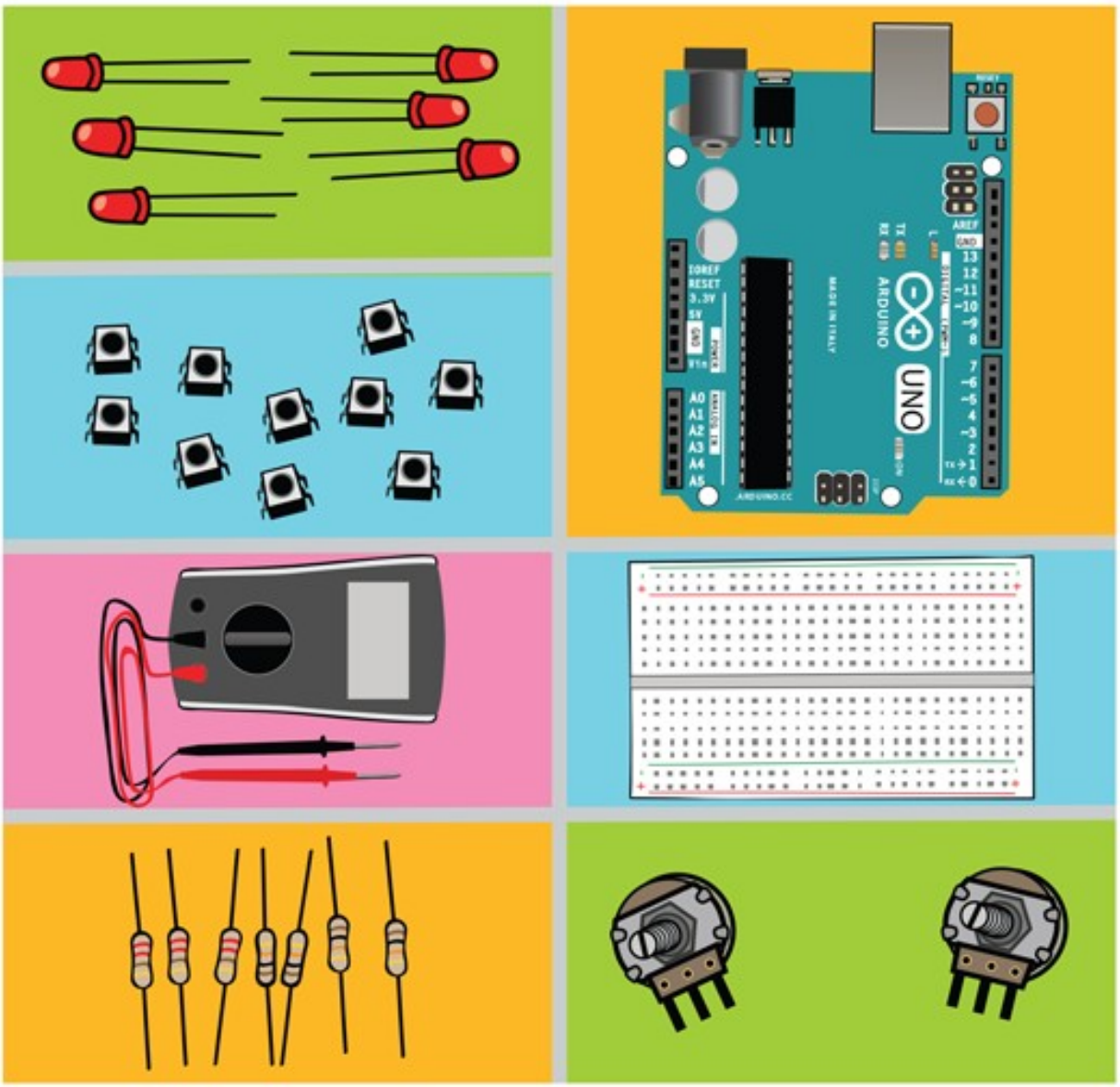
Make:

LEARN ELECTRONICS WITH ARDUINO



AN ILLUSTRATED BEGINNER'S GUIDE TO PHYSICAL COMPUTING

JODY CULKIN AND ERIC HAGAN

CONTENTS

Chapter 4: Programming the Arduino

Chapter 5: Electricity and Metering

Chapter 6: Switches, LEDs, and More

Chapter 7: Analog Values

Chapter 8: Servo Motors

Make:

LEARN ELECTRONICS WITH ARDUINO

AN ILLUSTRATED BEGINNER'S GUIDE TO PHYSICAL COMPUTING

JODY CULKIN AND ERIC HAGAN

MAKER MEDIA
SAN FRANCISCO, CA

Printed in the United States of America.

Published by Maker Media, Inc., 1700 Montgomery Street, Suite 240, San Francisco, CA 94111

Maker Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (safaribooksonline.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Publisher and Editor: Roger Stewart
Copy Editor: Elizabeth Welch, Happenstance Type-O-Rama
Proofreader: Scout Festa, Happenstance Type-O-Rama
Interior Designer, Compositor, and Cover Designer: Maureen Forys, Happenstance Type-O-Rama
Indexer: Valerie Perry, Happenstance Type-O-Rama

August 2017: First Edition

Revision History for the First Edition
28-08-2017 First Release

See oreilly.com/catalog/errata.csp?isbn=9781680453744 for release details.

others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

9-781-68045-374-4

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business. Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training. Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals. Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions to the publisher:
Maker Media
1700 Montgomery St.
Suite 240
San Francisco, CA 94111

You can send comments and questions to us by email at books@makermedia.com.

Maker Media unites, inspires, informs, and entertains a growing community of resourceful people who undertake amazing projects in their backyards, basements, and garages. Maker Media celebrates your right to tweak, hack, and bend any Technology to your will. The Maker Media audience continues to be a growing culture and community that believes in bettering ourselves, our environment, our educational system—our entire world. This is much more than an audience, it's a worldwide movement that Maker Media is leading. We call it the Maker Movement.

To learn more about Make: visit us at makezine.com. You can learn more about the company at the following websites:

Maker Media: makermedia.com
Maker Faire: makerfaire.com
Maker Shed: makershed.com

DEDICATION

Dedicated to all of our students, past, present, and future. Their curiosity drives them and inspires us.

ACKNOWLEDGMENTS

This book wouldn't have been possible without the help of many people, more than we can mention here. We'd like to thank our tech editor, Anna Pinkas, for her tireless and thorough review of this text. An earlier version of this book also benefited from tech editing by Michael Colombo and Sharon Cichelli. Roger Stewart, our publisher and editor, has been supportive and helpful throughout the process of getting this book into print. Our production team from Happenstance Type-O-Rama has been a delight to work with, particularly Liz Welch and Maureen Forys. We met at the Interactive Telecommunications Program at New York University, and we will always be grateful to Tom Igoe for suggesting we work together on a project there. In fact, we'd like to thank all of the faculty and staff at ITP, especially Dan O'Sullivan and Marianne Petit.

Eric would like to thank his wife Marie for her endless support, without which this book would not be possible. He would also like to thank his parents, David and Tracey, who have always had so much faith in his work.

Jody would like to thank her husband Calvin Reid, who seems to think she can do anything and has done whatever he can to make that possible. And she would like to acknowledge the memory of her parents, Florence and Hosmer Culkin, who would be startled but proud that she has co-authored a book on technology.

ABOUT THE AUTHORS

Jody Culkin is an artist and teacher. She has shown her sculptures, photographs, and installations at museums and galleries throughout this country and internationally. She illustrated *How to Use a Breadboard*, written by Sean Ragan, for Maker Media (2017). Her comic *Arduino!* has been translated into 12 languages. She has received grants and awards from the National Science Foundation, the New York State Council on the Arts, and many other organizations. She is currently a professor at City University of New York's Borough of Manhattan Community College in the Media Arts and Technology Department. She has a BA from Harvard University in visual studies and an MPS from NYU's Interactive Telecommunications Program.

Eric Hagan is an interactive and kinetic artist and professor based out of Astoria, New York. He has written articles for publications, including *Make:* magazine and *Popular Science*. He has also worked on several art installation projects around New York City, including the annual holiday windows on 5th Avenue and Kara Walker's *A Subtlety*. He is currently an assistant professor at SUNY Old Westbury in the Visual Arts Department. He has a BA from Duke University in philosophy and an MPS from NYU's Interactive Telecommunications Program. Eric enjoys showing projects at the annual New York City World Maker Faire.

PREFACE

We conceived of this book as an introduction to electronics and the Arduino platform for the complete beginner. We have written and illustrated it assuming that the reader has no prior knowledge of either electronics or programming. As the reader progresses through the book, electronics and programming concepts are thoroughly explained, in text and with images. After the reader has completed the book, they will be able to use it as a reference for basic electronics and Arduino programming.

This book should be the jumping-off point for creative projects. When finished reading the book and completing all the exercises in it, readers should be equipped to start developing their own projects. We haven't covered everything that the Arduino can do, but we have set readers on their way to finding that out for themselves.

Many of the code sketches used in this book are taken from the examples in the Arduino IDE. The other sketches are available here: github.com/arduinotogo/LEA.

1

INTRODUCTION TO ARDUINO

Perhaps you have seen the Arduino at a local retailer, heard about it from a friend who purchased one, or just saw a cool project on the Internet that piqued your interest. What is the Arduino? Most simply, it is an affordable, small-scale, simple computer that focuses on interaction with the outside world (Figure 1-1).

Most of the computers you are familiar with are controlled almost exclusively through the keyboard and mouse, touchscreen, or trackpad. An Arduino allows you to take information from the outside world with sensors that measure temperature, light and sound levels, or even the vibrations underneath your feet, and convert these measurements into motion, sound, light, and more.

FIGURE 1-1: The Arduino logo

The Arduino was originally developed by teachers to make it possible for their design students who were not engineers to create interactive objects and environments. Since the original Arduino was released in 2005, it is estimated that over 1 million have been sold. Designers, educators, engineers, hobbyists, and students have built all kinds of projects that sense and respond to the world with Arduino.

There are many versions of the Arduino, and each is designed for a specific function. Figure 1-2 shows a few of the Arduino boards.

We have written this book in the spirit of the Arduino team. We don't assume that you already know programming or electronics—we will show you what you need to know to get up and running with the Arduino. It will help if you are good at building and tinkering, and you have a determined nature.

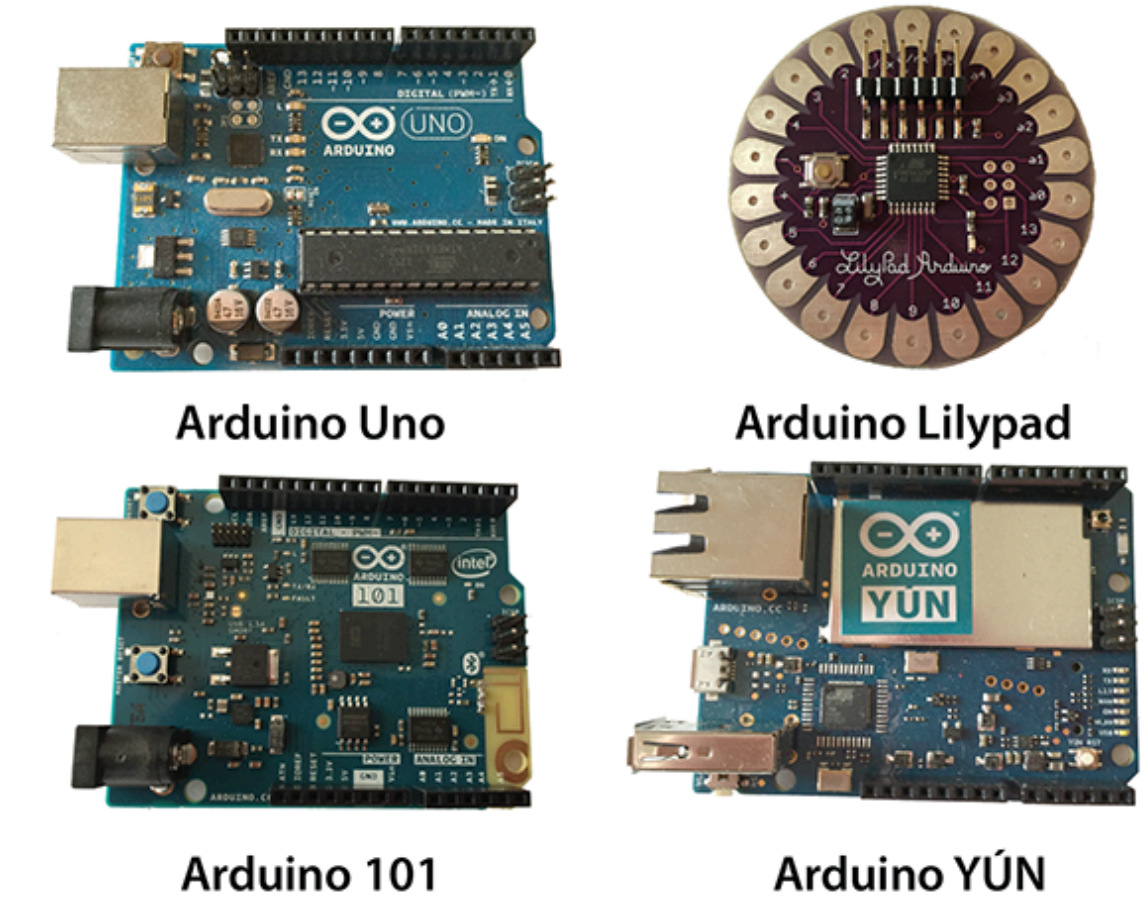


FIGURE 1-2: There are many versions of the Arduino, each designed for a different function.

PHYSICAL COMPUTING

The Arduino is used for building physical computing projects. What does that mean? Physical computing refers to taking information from the world around us by using inputs such as sensors and switches and responding to that information with outputs of some kind. It could be as simple as turning on an LED when a room gets dark, or it could be a complex system of sound and light that responds to the position of a person in a room. An Arduino can act as the "brains" of this kind of a system, handling the information coming in and the response going out.

The Arduino is part of the open source hardware movement. Let's look at what that means.

WHAT IS OPEN SOURCE HARDWARE?

The Arduino is defined on its website as an open source electronics prototyping platform. In the open source hardware movement, technologists share their hardware and software to foster development of new projects and ideas. Source designs are shared in a format that can be modified, and whenever possible, readily available materials and open source tools are used to create the designs.

By encouraging the sharing of resources, the open source hardware movement facilitates development of new products and designs. Open source projects emphasize the importance of documentation and sharing, making the community of users a great resource for learners.

PROTOTYPING

The Arduino is a prototyping platform. What's prototyping? It is building a model of a system. It can involve many phases, from initial sketches through detailed plans and a series of refinements, to building a fully functional model that can be replicated. Or it can be a quick one-off that's put together rapidly to test an idea.

WHAT WILL I NEED AND WHERE CAN I GET IT?

There are several versions of the Arduino; it has been around since 2005 and is constantly evolving. For the purpose of this book, we are concerned with the Arduino Uno. Your Arduino might not look exactly like the Uno shown in Figure 1-3, because we have simplified the drawing in order to point out the sections that concern us. Since the Arduino is open source, you might also purchase a board that does

not come directly from the Arduino organization. Just know that for this book we are focused on the Arduino Uno and compatible boards.

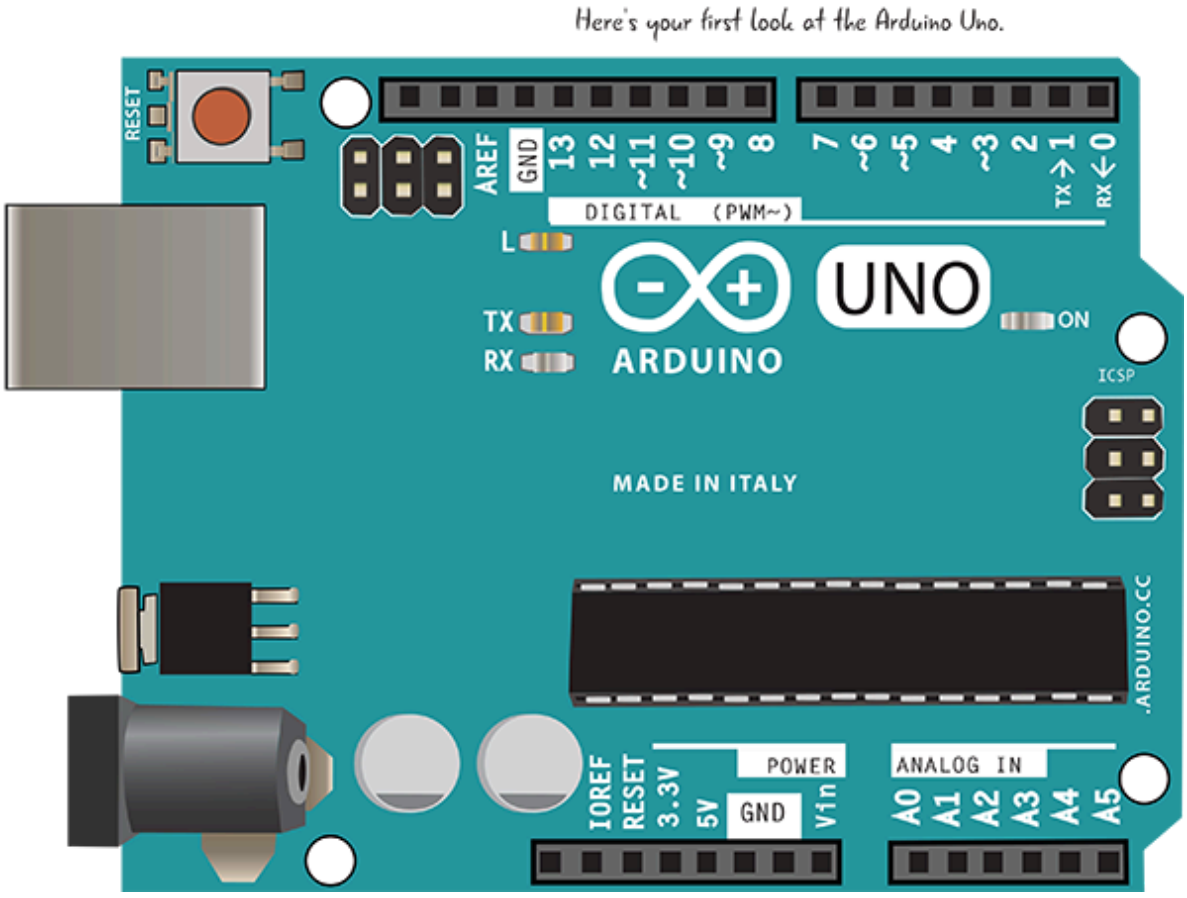


FIGURE 1-3: The Arduino Uno

PARTS AND TOOLS

We will also need some additional electronic parts and a few tools to build projects with the Arduino. Here is a list of the parts you will need to purchase to complete the projects in this book. We'll give you more detail about the parts and what they do as we build each project.

PARTS LIST

Breadboard

USB A-B cable

9-volt battery

9–12-volt power supply

9-volt battery cap or holder

Assorted LEDs, a variety of colors

Assorted resistors

10K potentiometer

3 momentary switches/buttons

Photoresistor

Speaker, 8 ohm

2 servo motors

Jumper wires

The next few figures, Figure 1-4 through Figure 1-16, show you what the parts look like, along with a brief description. Electronic parts are often called components, because they are components in an electronic circuit. You'll learn more about circuits in Chapter 3, "Meet the Circuit."

A breadboard, shown in Figure 1-4, is used to build and test circuits quickly. A USB A-B cable, shown in Figure 1-5, connects the Arduino to a computer so you can program it. It will also provide power. A 9-volt battery, shown in Figure 1-6, can provide power when the Arduino is not attached to a computer.

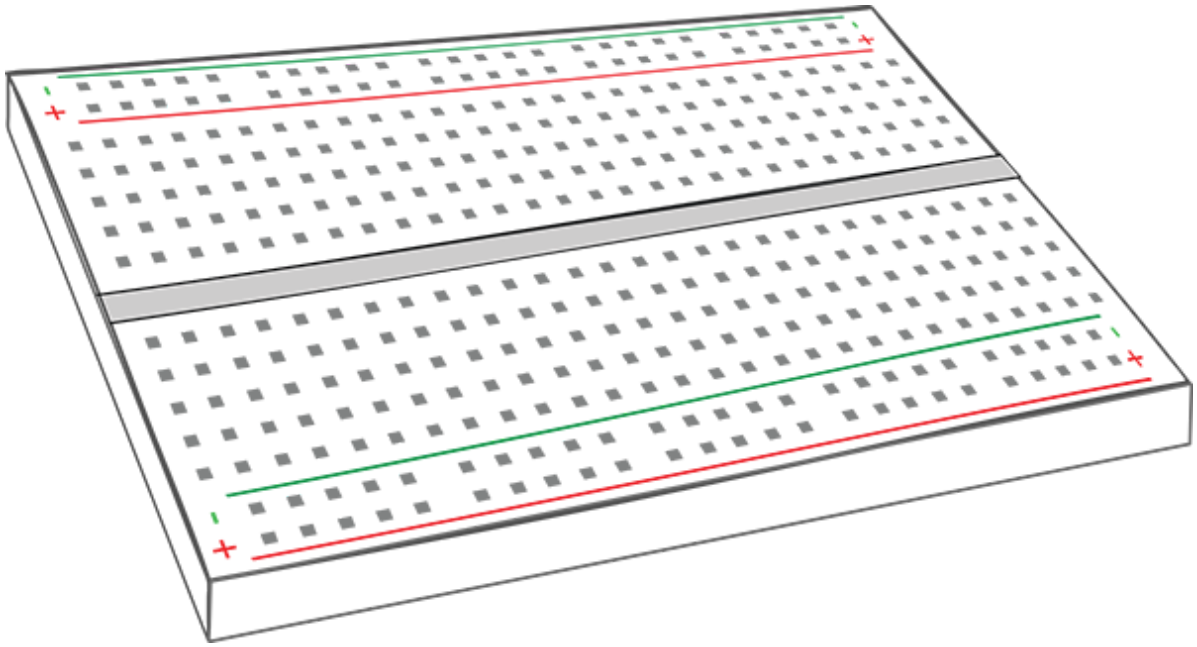


FIGURE 1-4: Breadboard

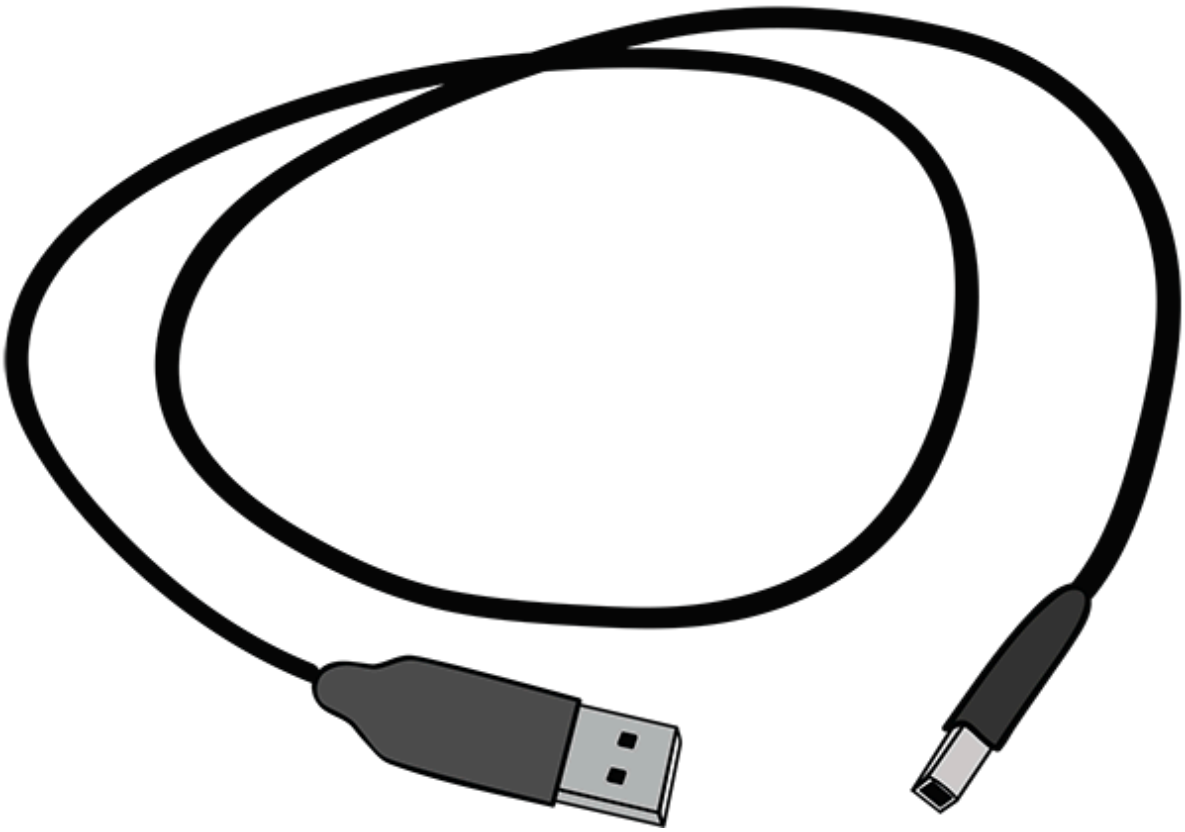


FIGURE 1-5: USB A-B cable

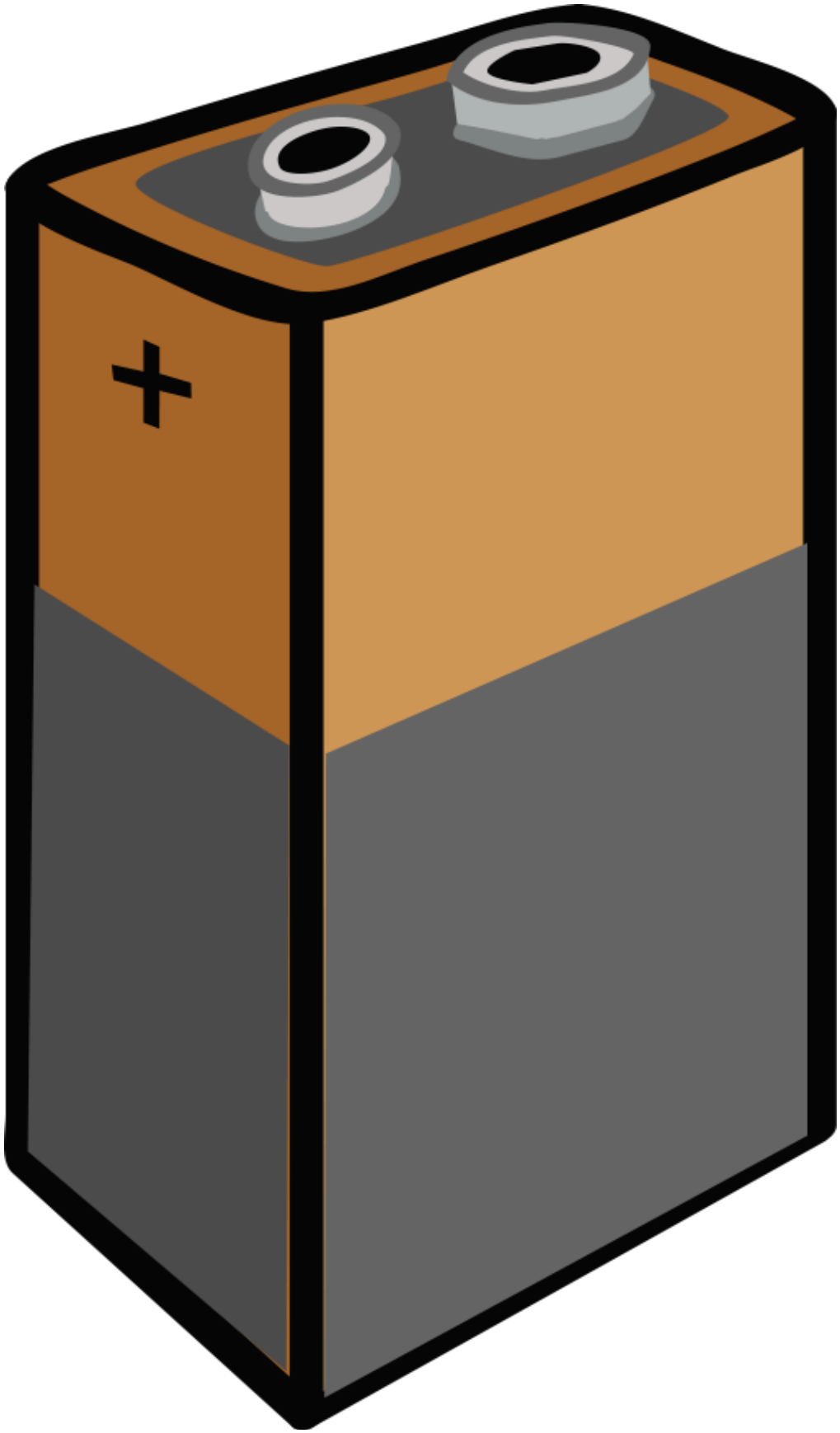


FIGURE 1-6: 9-volt battery

FIGURE 1-7: Battery cap

The battery cap, shown in Figure 1-7, will be used to attach a battery to a breadboard. The power adapter, shown in Figure 1-8, can power your Arduino when it is not attached to your computer. Light-emitting diodes (LEDs), shown in Figure 1-9, emit light when a voltage is applied.

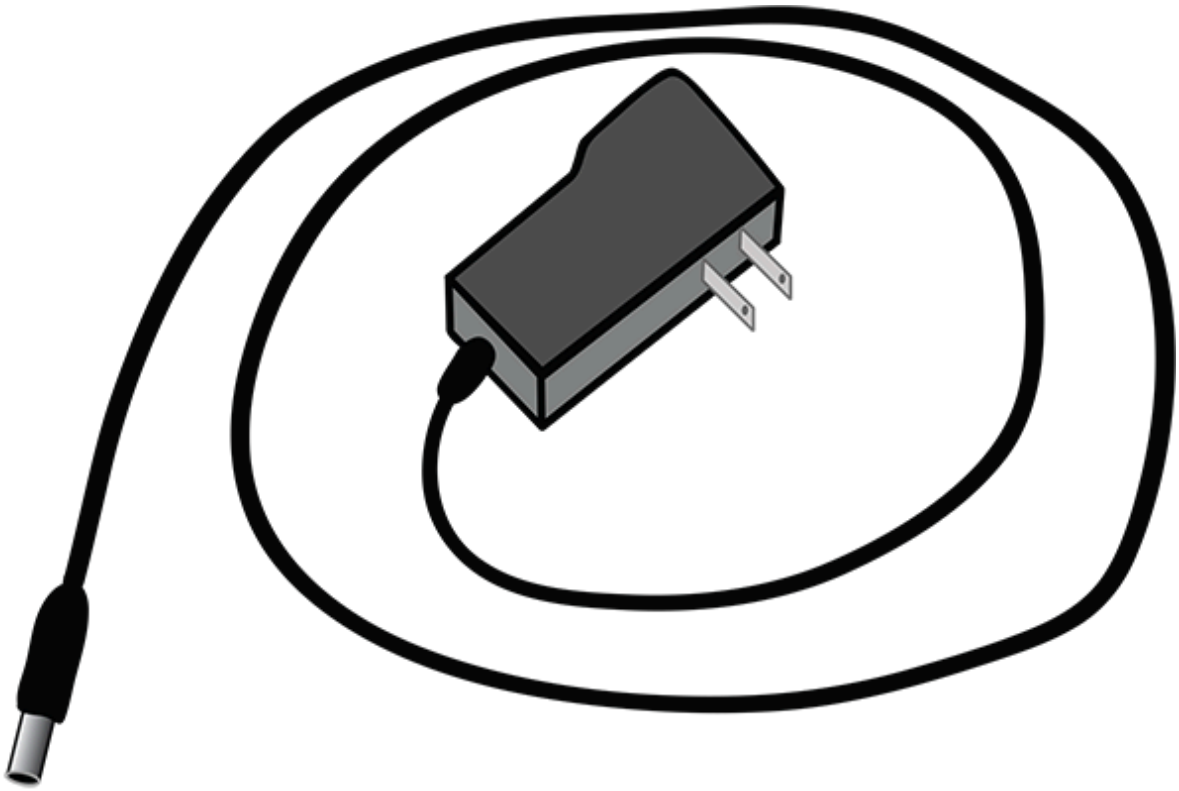


FIGURE 1-8: Power adapter

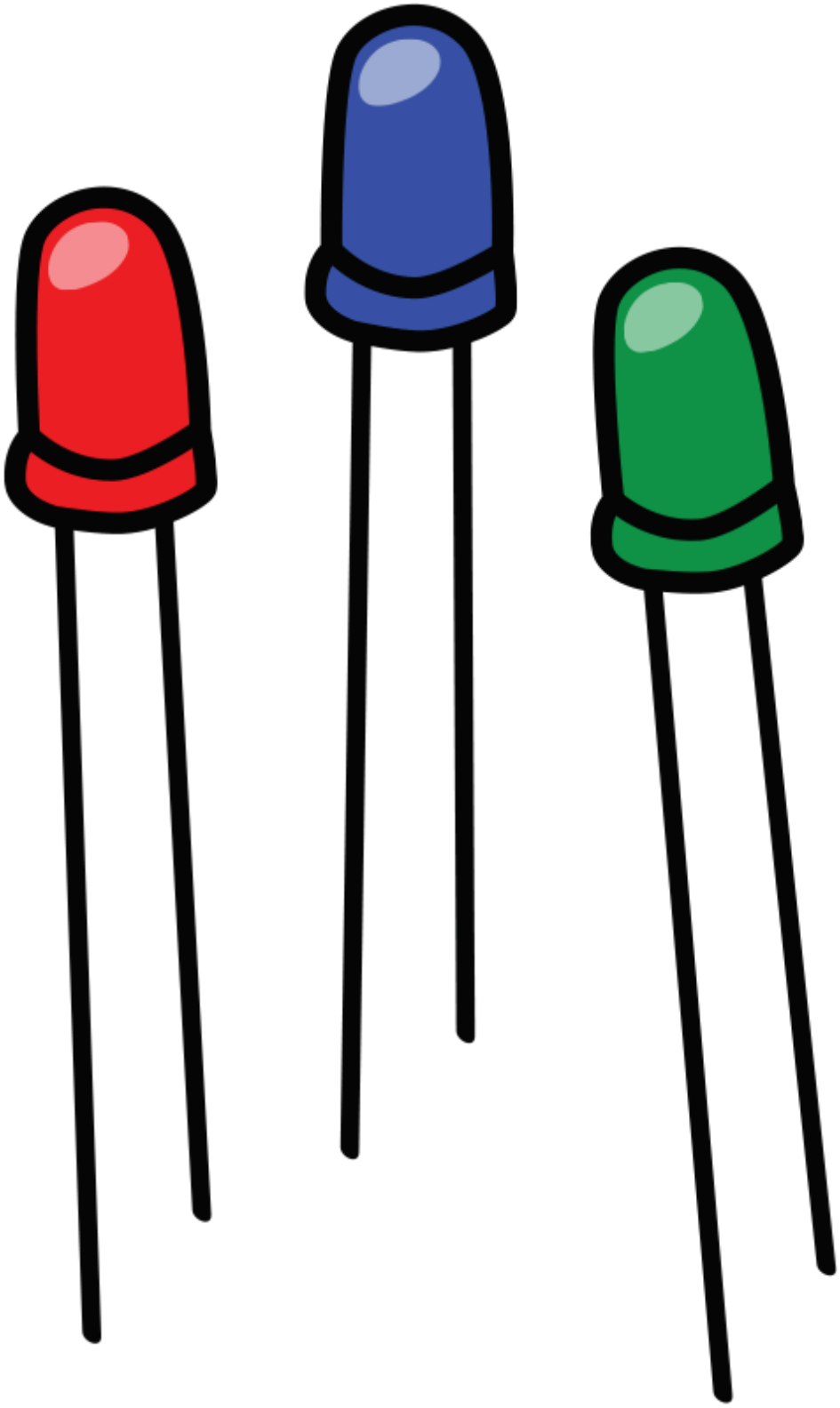


FIGURE 1-9: LEDs

Resistors, as you can see in Figure 1-10, limit the flow of current in a circuit. We will use a momentary pushbutton, shown in Figure 1-11, to make or break a connection in a circuit. Figure 1-12 shows a potentiometer, a variable resistor.

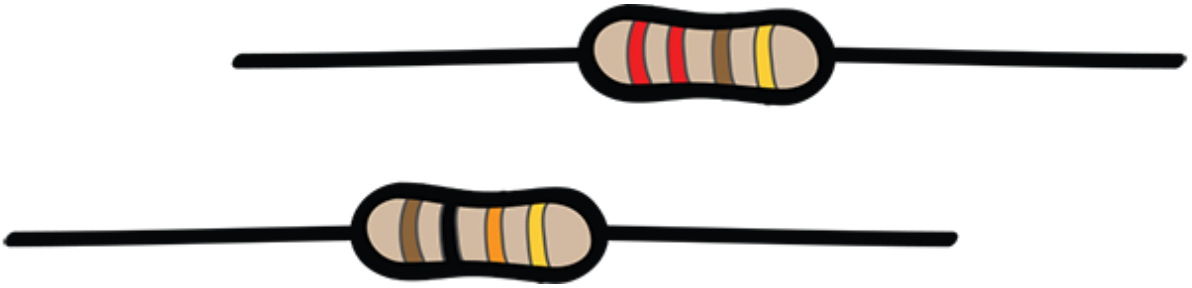


FIGURE 1-10: Resistors

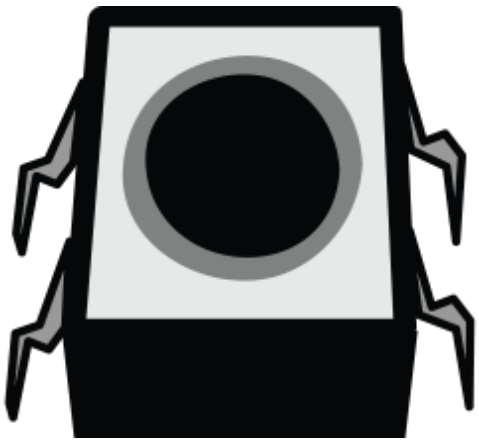


FIGURE 1-11: Momentary pushbutton

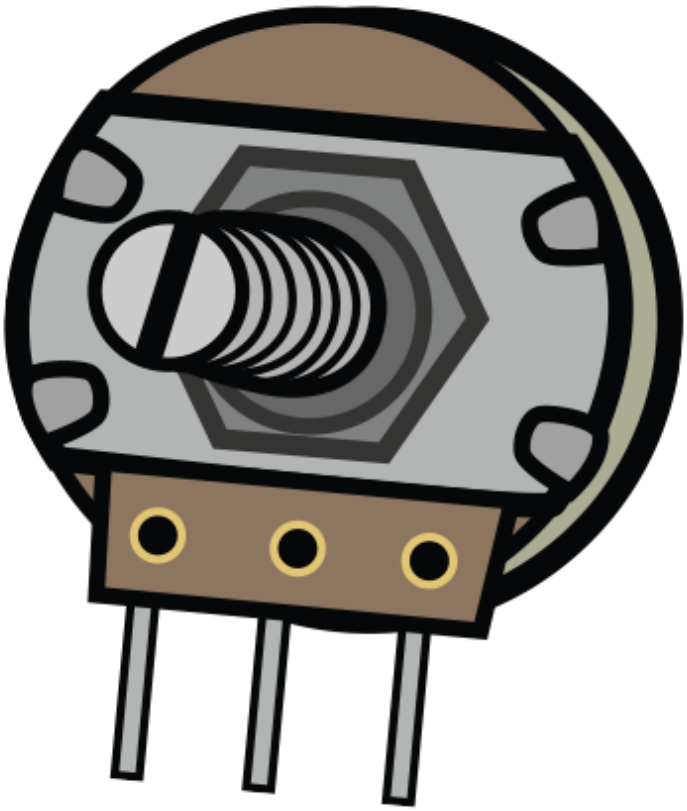


FIGURE 1-12: Potentiometer

A photoresistor, shown in Figure 1-13, changes its resistance when exposed to different levels of light. Figure 1-14 shows an 8-ohm speaker, which will play audio signals. The servo motor is an easily controlled hobby motor, as you can see in Figure 1-15. Jumper wires, shown in Figure 1-16, are used to connect components in a breadboard. You can buy them or make them yourselves with wire strippers.

FIGURE 1-13: Photoresistor

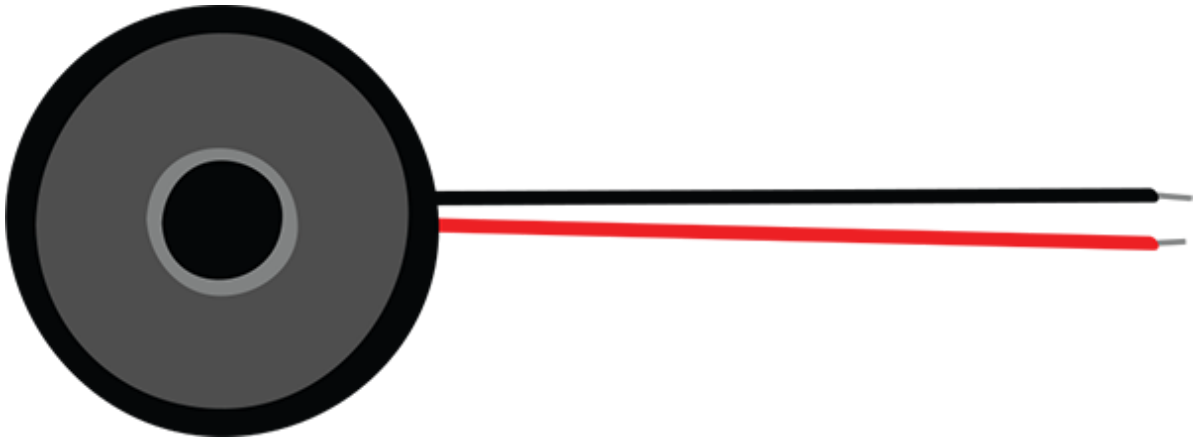


FIGURE 1-14: Speaker, 8 ohm

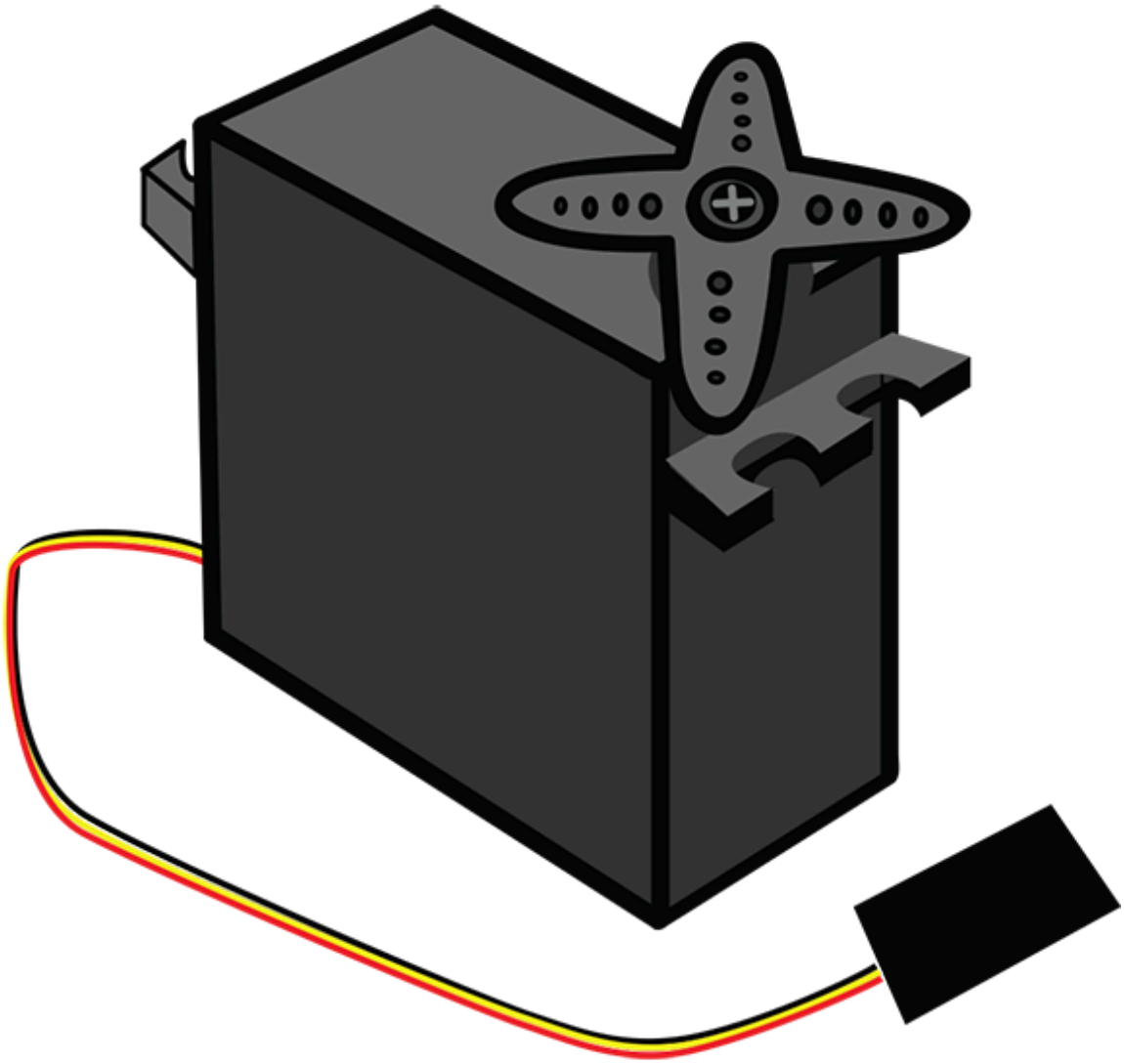


FIGURE 1-15: Servo motor

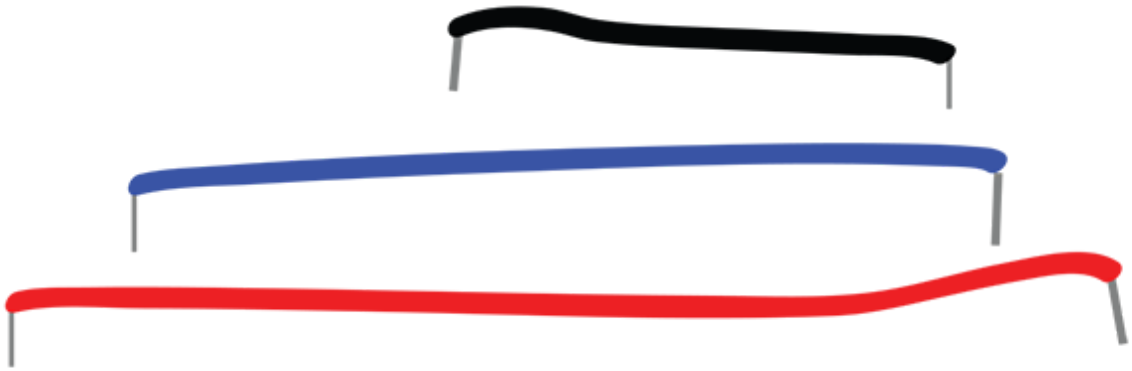


FIGURE 1-16: Jumper wires

A NOTE ABOUT LEDS

LEDs come in a variety of colors, styles, and sizes. We will use LEDs in many of the projects in this book because they help demonstrate a number of basic electronics and Arduino concepts in a visual way.

One important thing to remember about LEDs is that they have a polarity, or direction in which they must be placed in order to work in a project. If we place the LEDs backward, they won't light up. How do we know the orientation of an LED?

LEDs have two legs, or leads, which are different lengths, as you can see in Figure 1-17. The longer lead is known as the anode, the side of the LED that we will connect to power. The shorter leg is called the cathode, which will be pointed away from our power source. We'll show you how to position the leads in a circuit when we start building one, and we'll always remind you of the polarity in later circuits.

Note

If you place the LED in backward, it won't light up but it also won't damage anything in your project.

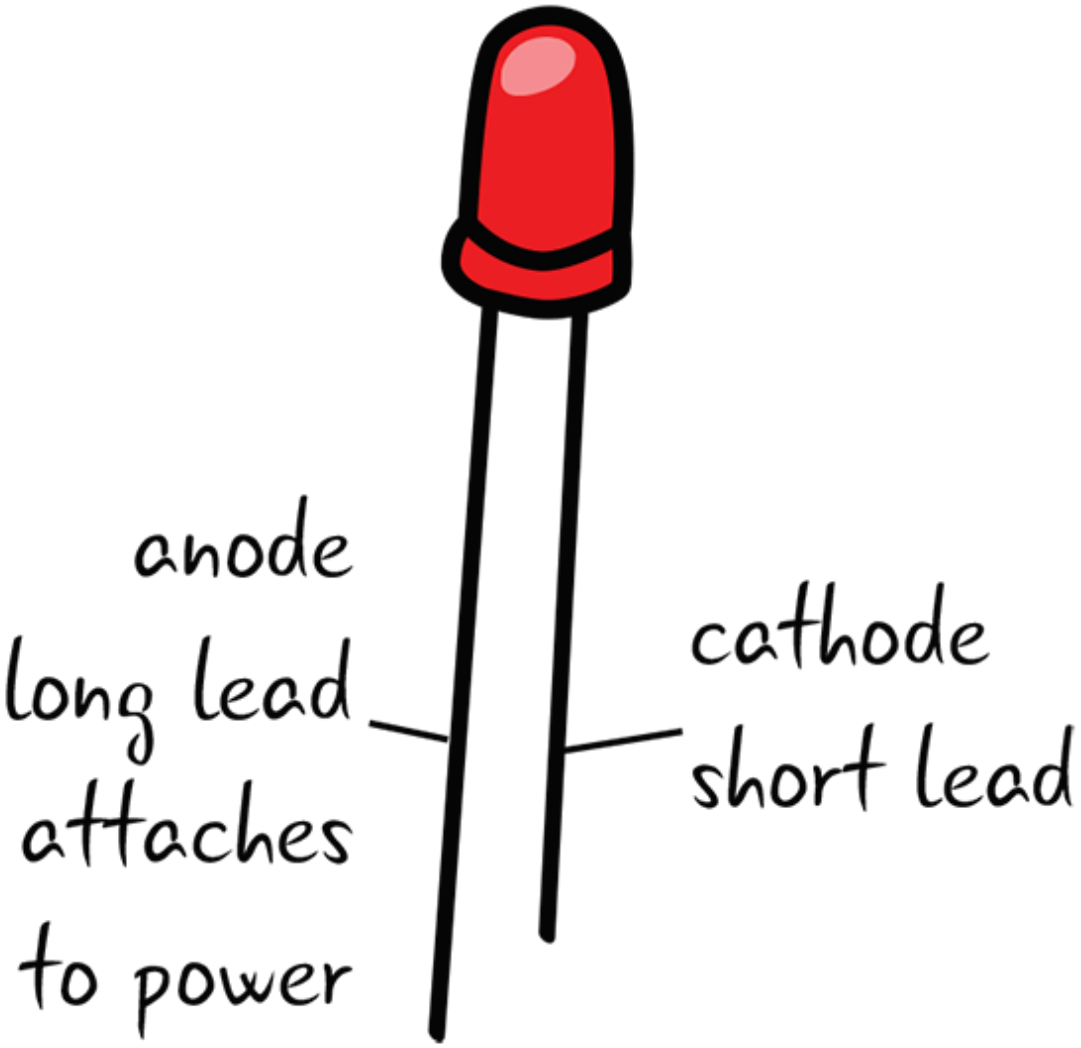


FIGURE 1-17: Anode (positive lead) and cathode (negative lead) of an LED

What happens if you have a used LED that has clipped leads? In many LEDs, if you feel the bulb, one side of the rim at the bottom of the bulb feels flatter. The lead connected to that side is the cathode, or negative side.

Now let's take a look at a few tools you will need to make these projects.

TOOLS

A multimeter will tell you everything you need to know about the electrical properties of a circuit, properties that are not necessarily visible to your eye. We will show you how to use it, starting in Chapter 2. The multimeter depicted in Figure 1-18 is available from SparkFun (part number TOL-12966), but you may find another one that you like. When you choose a multimeter, make sure it is digital and has removable leads, and that it is fused.

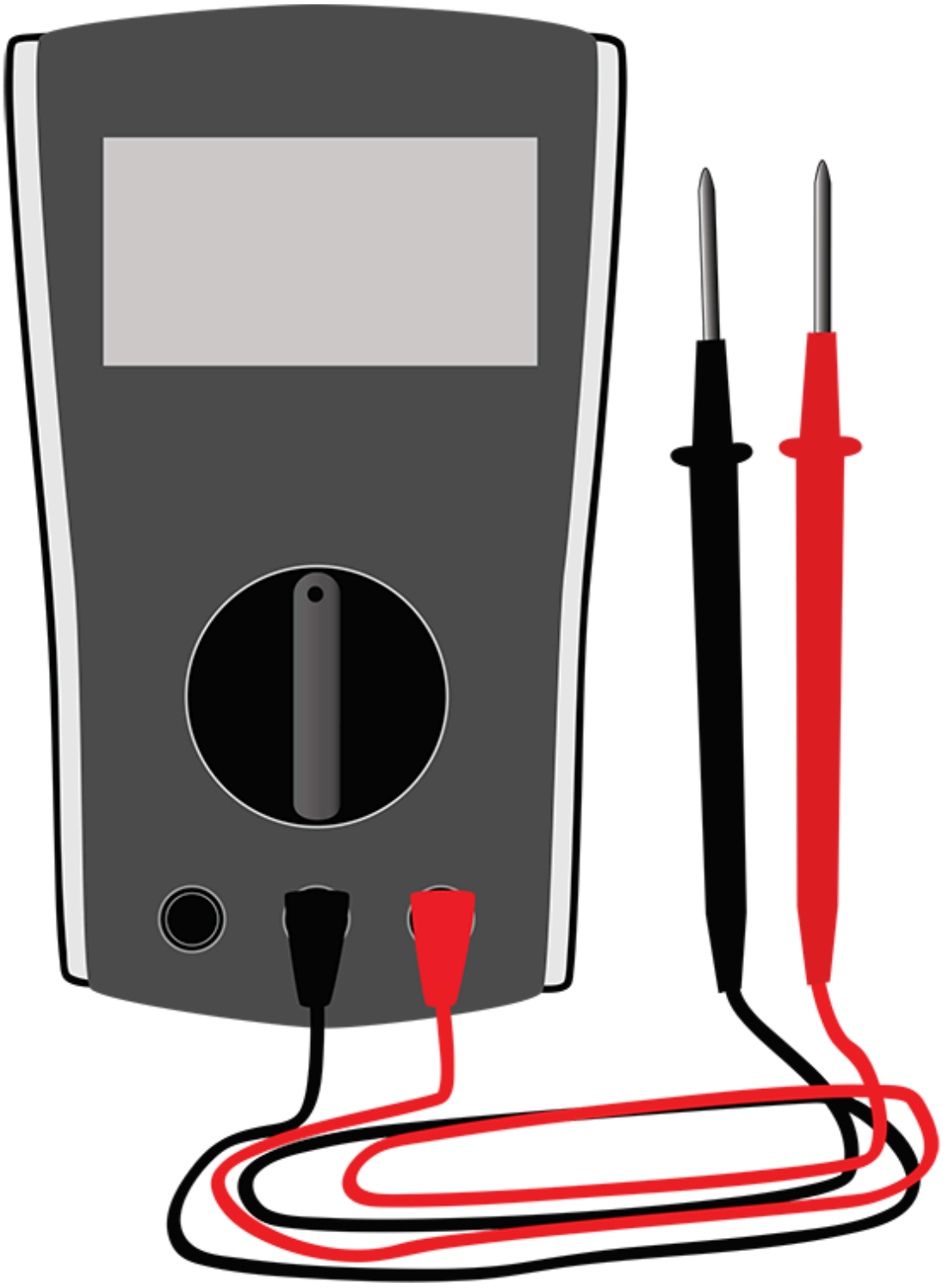


FIGURE 1-18: Multimeter

Needle-nose pliers, as shown in Figure 1-19, come in handy for pulling components out of the breadboard when you wish to make

changes to a circuit. They are also helpful for picking up small components.

Wire strippers, pictured in Figure 1-20, are used to pull off the plastic insulating coating found on various thicknesses of wire. They will make your life a lot easier when using spools of wire, since you will be able to cut and use custom lengths of wire.

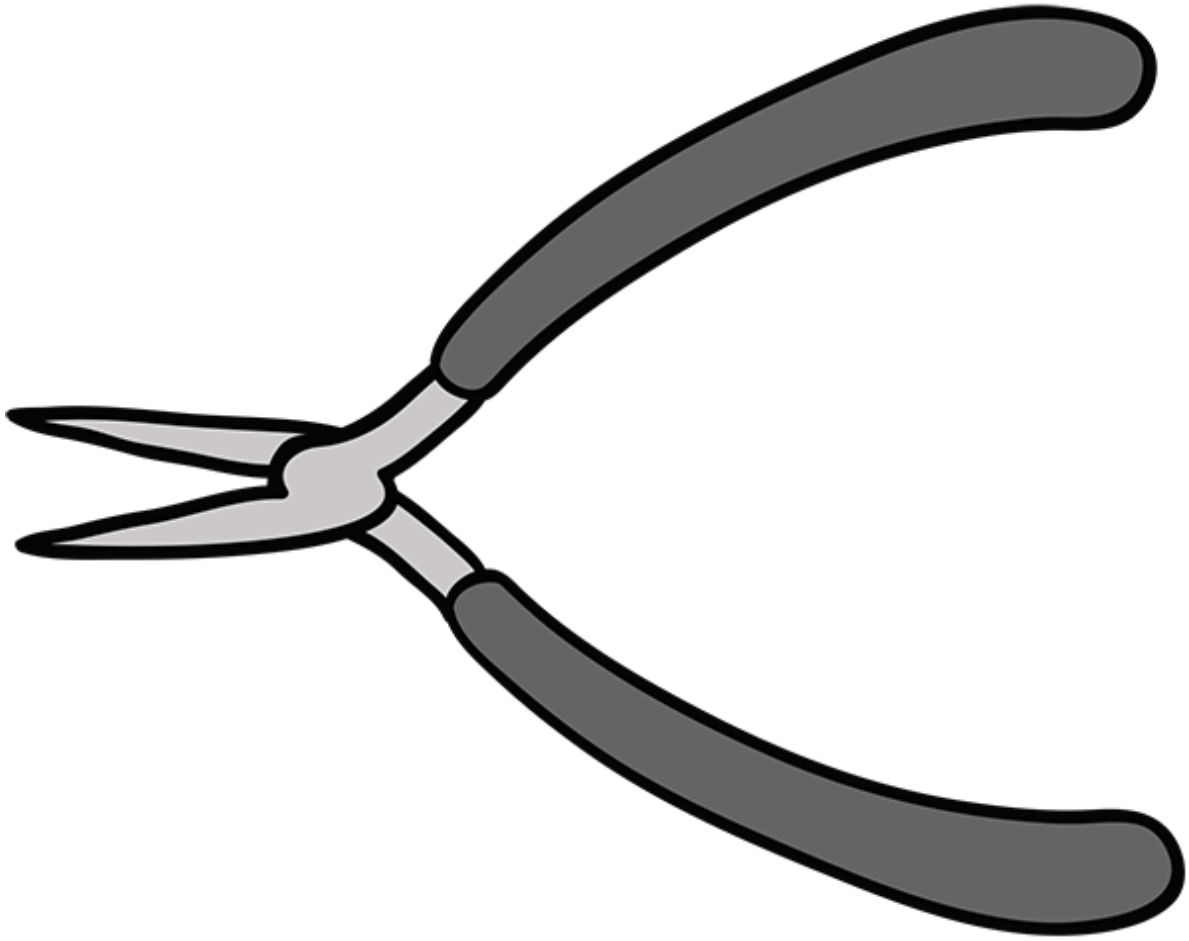


FIGURE 1-19: Needle-nose pliers

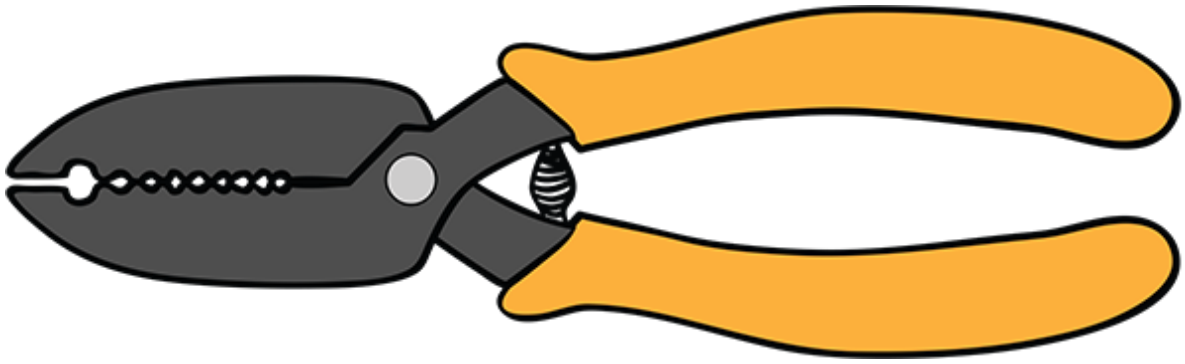


FIGURE 1-20: Wire strippers

> **Tip**
>
> Although you can buy precut jumper wires, remember that you can create your own by using your wire strippers to strip off the plastic coating on the ends of a segment of wire. Twenty-two-gauge hookup wire works well in breadboards.

A WORD ABOUT TOOLS: THE SOLDERING IRON

You may be familiar with a soldering iron and its use in electronics to connect components. In this book we have elected to use a breadboard to make connections in all the circuits listed. This means that you are not required to purchase a soldering iron or learn how to use one to complete the projects in this book.

QUESTIONS?

Q: What does a soldering iron do?

A: A soldering iron is used to melt a conductive material ("solder") to combine electrical components in a permanent way. This process is called *soldering*.

Q: Why aren't you teaching soldering in this book?

A: Soldering is a wonderful skill to have and will help you take your electronics to the next level, but for this book we were primarily concerned with the basics. You can make fully functional circuits without it.

Q: The list of components seems to have a lot of parts to it. The pictures look nice, but do I really need to purchase all the items in that list?

A: You will be seeing a lot more of those pictures! To answer your question, you will be using all of those parts when you build the projects in this book. These parts can also be reused for your own projects. We will explain what all of these parts do as we use them.

Q: My friend/sibling/parent/teacher/dog gave me a newer/older model of the Arduino. Do I have to use the Arduino Uno for the projects in this book?

A: Good question. The projects in the book might work with your particular Arduino, but both the programming and the abilities of the Arduino have changed over time *and* differ based on the version. All of the examples in this book have been tested using the Arduino Uno and the latest release of the Arduino software.

Q: I don't recognize or know how to use any of the tools or components you have shown; is there another book for me?

A: No! This book *is* written for you. We will be covering specifics on how to use all of the parts and tools we have listed in the coming chapters. Sit tight and keep reading.

Q: I don't have anywhere in my neighborhood to purchase those parts. Do you have any recommendations for places I can find those parts online?

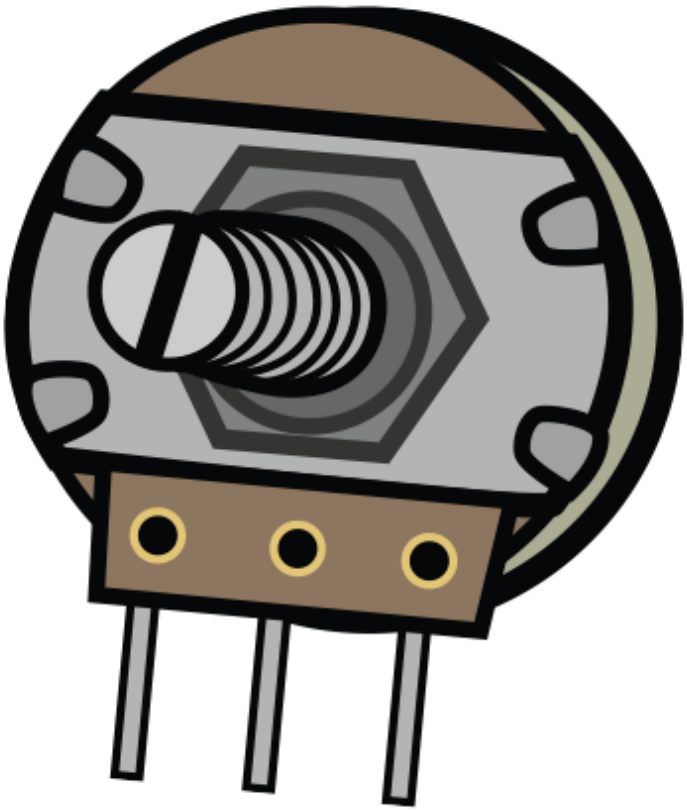
A: Great question! You are ready for the next section.

RESOURCES

A number of vendors sell the components that you will need. Here are the URLs of the websites of many of them, and there may be brick-and-mortar stores or other resources in your community.

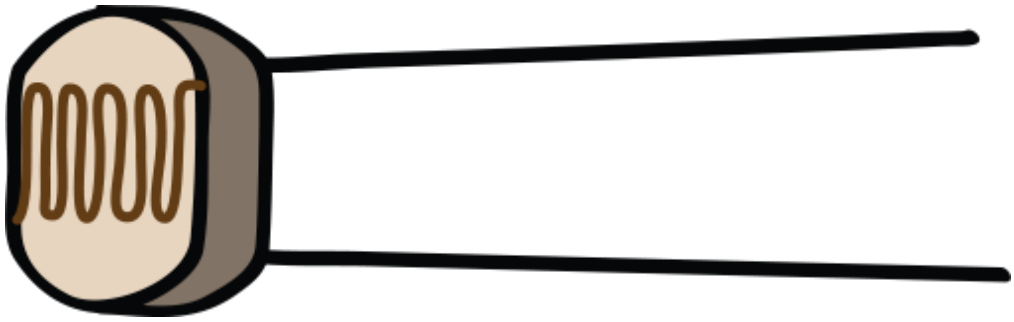
Maker Shed (makershed.com)

Selection of kits and individual Arduino components. Some electronic parts, focused on the Maker community.



SparkFun Electronics (sparkfun.com)

Wide range of sensors and breakout boards, classic Arduinos and their homemade version.

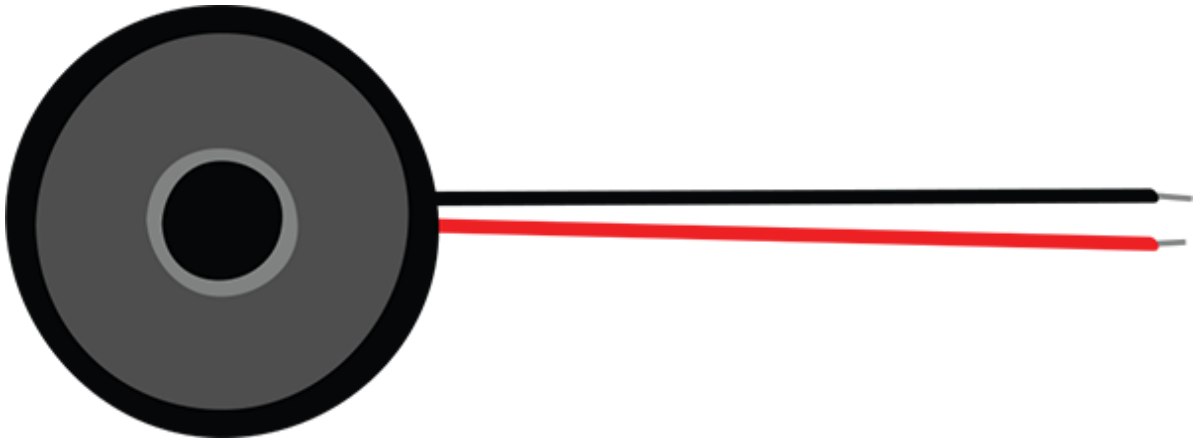


Adafruit Industries (adafruit.com)

Arduinos and breakout boards, sensors, electronic components.

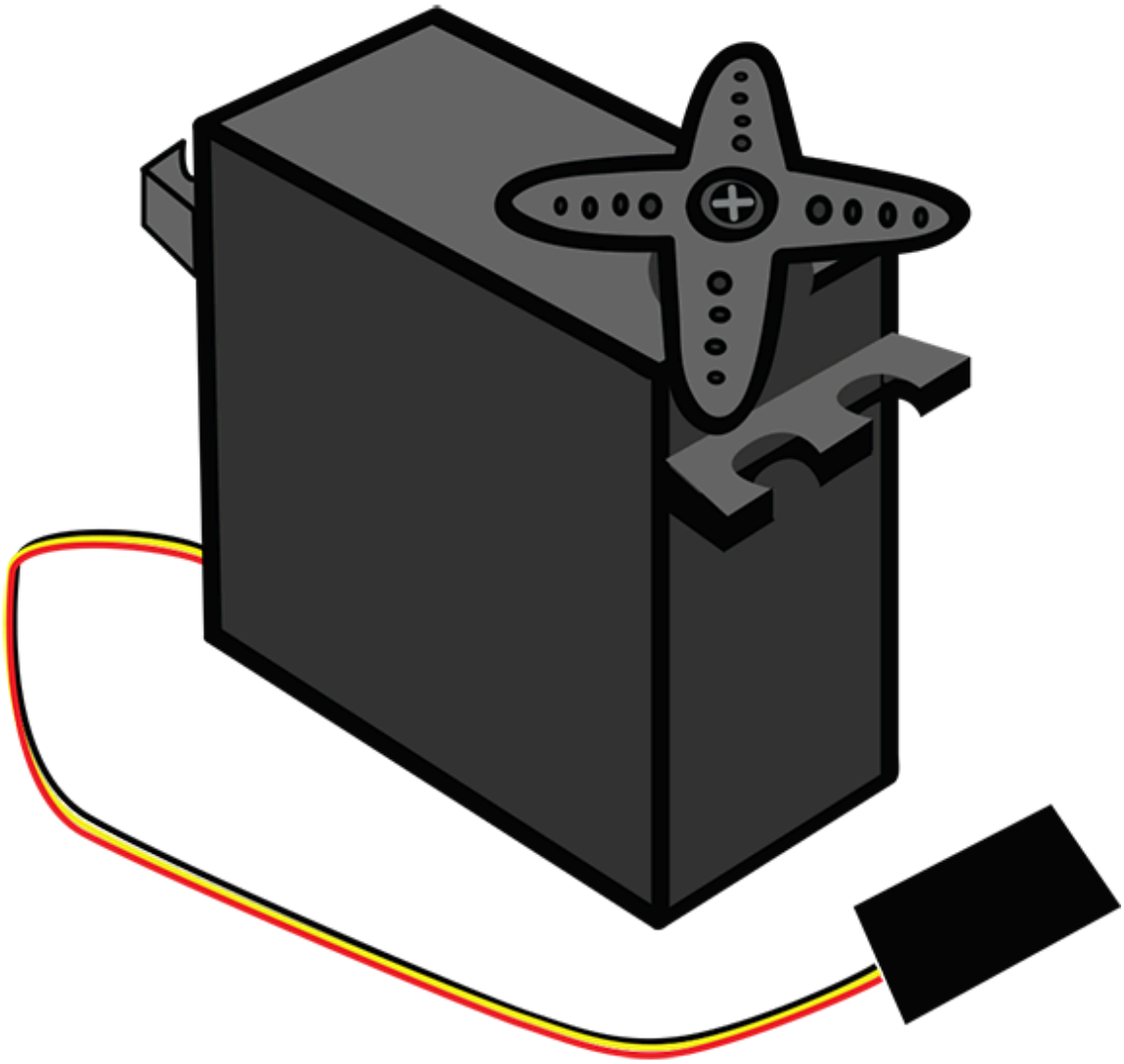
Jameco Electronics (jameco.com)

Mostly electronics components, endless buttons and switches.



Mouser Electronics (mouser.com)

Some Arduino, tons of electronics, sensors, and other items.

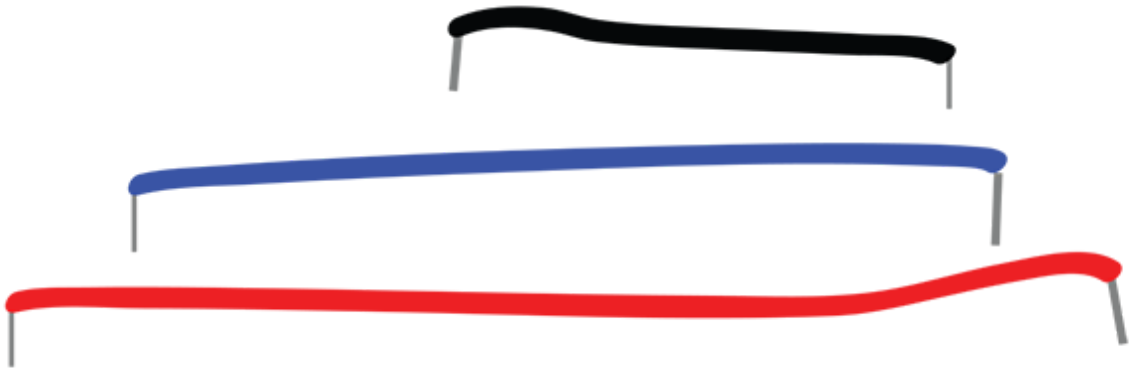


Digi-Key Electronics (digikey.com)

Great for ordering components, chips, and so on.

Micro Center (microcenter.com)

A source of components and Arduinos, they have some brick-and mortar-stores as well as a website.



KITS

Kits are available from some of the vendors mentioned here that have most of the parts you will need to complete the projects. We will review exactly what you need to build the projects in every chapter. Here are a few of the kits available; you will find that there are many more.

A kit developed by the Arduino team (arduino.cc/en/Main/ArduinoStarterKit). It can be purchased from a number of vendors.

This kit is available from the Maker Shed: makershed.com/products/make-getting-started-with-arduino-kit-special-edition

Adafruit Industries has a few kits, including this one: adafruit.com/products/193

SUMMARY

This chapter set you on the path to using your Arduino. By now you know where to get the required items, you can identify various components and tools you will use, and you know something about the contributions of the open source movement.

The next chapter will look at the Arduino Uno in more detail and show you how to hook it up to your computer.

2

YOUR ARDUINO

Now that you've got your Arduino and a number of parts and tools, let's look at them in more depth. The Arduino is just the thing to solve your everyday interactive needs. In this chapter, you'll learn about the parts of the Arduino and how to attach it to a computer and to a power supply. We will also look at unboxing our electronic parts, sorting them out, and learning more about them on both websites and data sheets.

PARTS OF AN ARDUINO

First let's take a look at the labeled parts of the board, as shown in Figure 2-1.

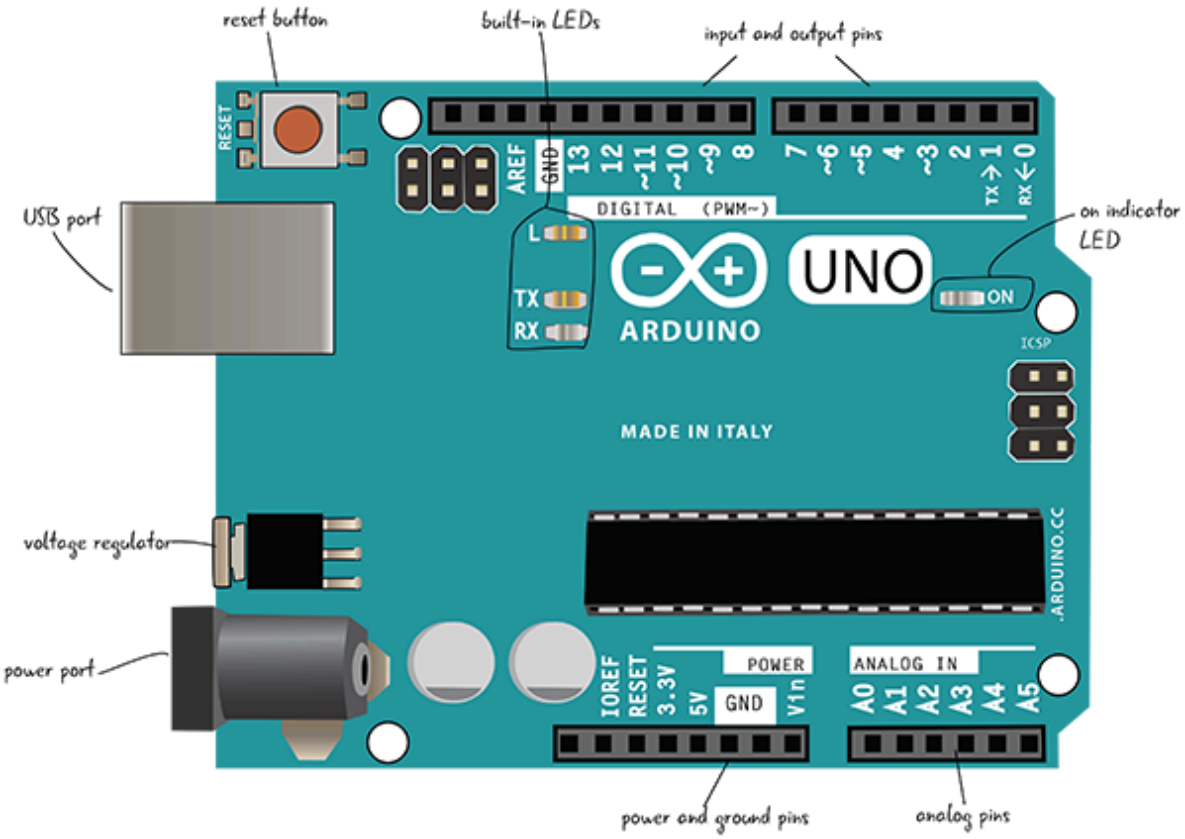


FIGURE 2-1: The Arduino Uno

We are going to break down each side of the board in more detail so you can see where everything important is located on the Arduino.

ARDUINO IN DETAIL

Let's learn a bit more about what is on the Arduino board. Remember that there are different styles of boards, so yours may look slightly different. These figures are based on Arduino Uno revision 3. We'll look first at the left side of the board, with the reset button, USB port, voltage regulator, and power port, as shown in Figure 2-2.

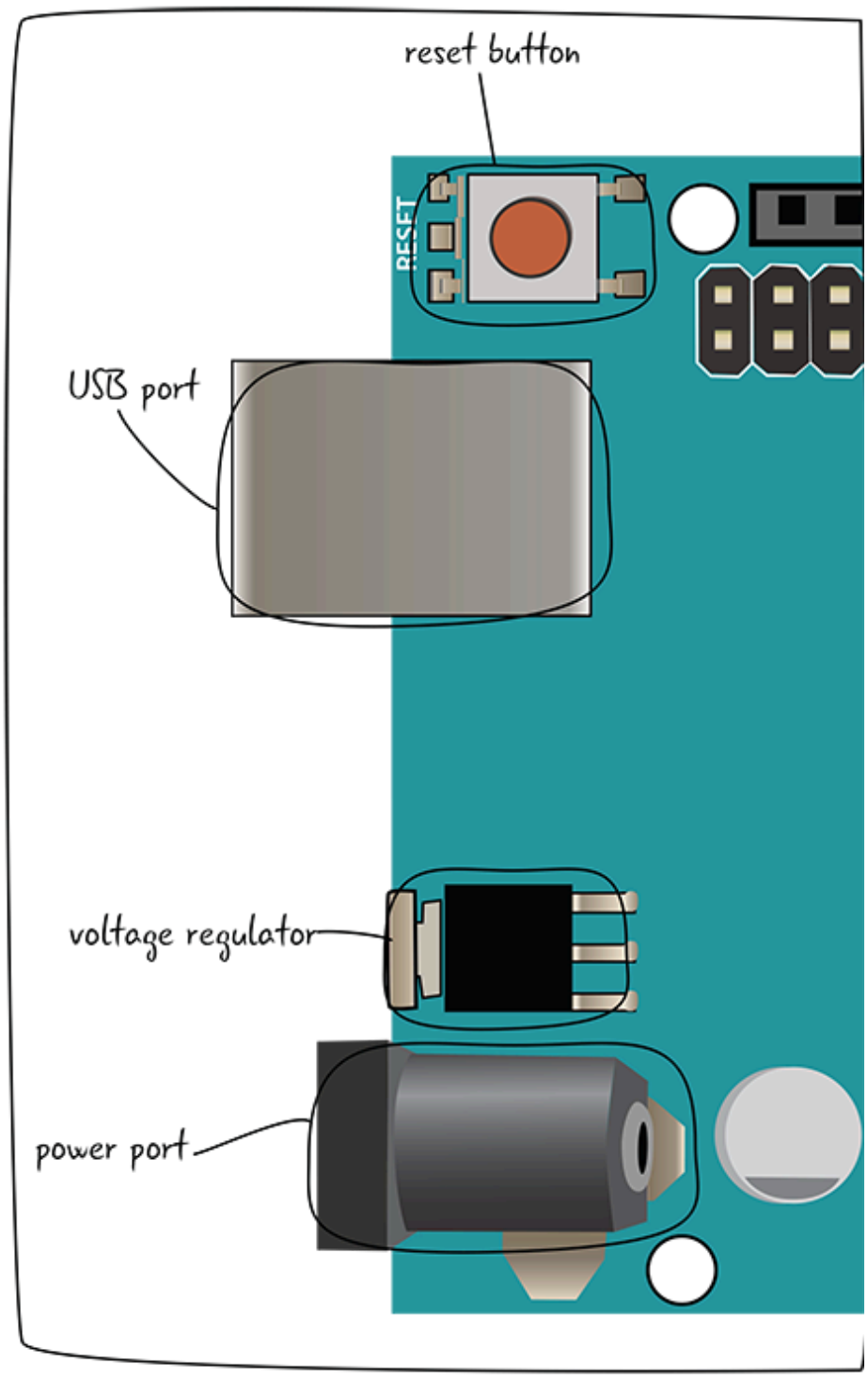


FIGURE 2-2: The left side of the Arduino Uno board

Reset Button

Much like turning your computer off and on again, some problems with the Arduino can be solved by pushing the reset button. This button will restart the code currently uploaded on your Arduino. The reset button may be in a different location on your board than in Figure 2-2, but it is the only button.

USB Port

The USB port takes a standard A-to-B USB cable, often seen on printers or other computer peripherals. The USB port serves two purposes: First, it is the cable connection to a computer that allows you to program the board. Second, the USB cord will provide power for the Arduino if you're not using the power port.

Voltage Regulator

The voltage regulator converts power plugged into the power port into the 5 volts and 1 amp standard used by the Arduino. *Be careful!* This component gets very hot.

Power Port

The power port includes a barrel-style connector that connects to power straight from a wall source (often called a wall-wart) or from a battery. This power is used instead of the USB cable. The Arduino can take a wide range of voltages (5V–0V DC) but will be damaged if power higher than that is connected.

We'll take a closer look at the other side of the board now (Figure 2-3), which includes the digital, analog, and power pins as well as the actual chip for the board.

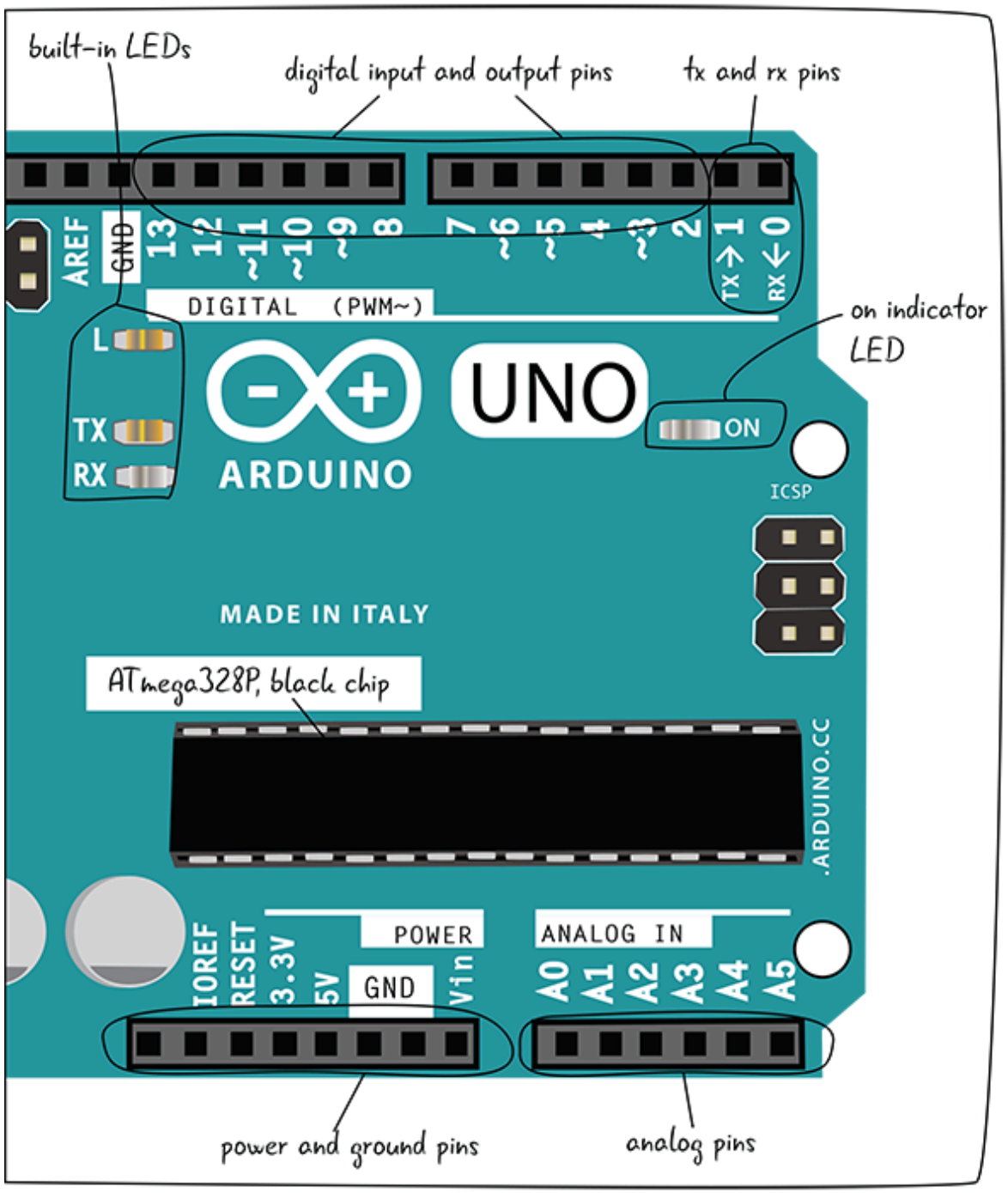


FIGURE 2-3: The right side of the Arduino Uno

Built-In LEDs

The LEDS marked TX and RX show whether your Arduino is sending or receiving data. The one marked L is connected to Pin 13.

ON Indicator LED

This LED indicates that the Arduino is getting power when you turn it on.

Digital I/O Pins

The holes on this side of the board are called the digital input/output pins. They are used to either sense the outside world (input) or control lights, sounds, or motors (output).

TX/RX Pins

Pin 0 and Pin 1 are special pins labeled TX and RX. We will cover this in more detail later, but it is good practice to leave these pins empty. Any changes you make to your program won't load if something is plugged into Pin 0.

ATmega328P, Black Chip

The black chip in the middle of the board is an ATmega328P. This is the "brains" of the Arduino: it interprets both the inputs/outputs and the programming code uploaded onto your Arduino. The other components on the board enable you to communicate with this chip when creating projects.

Power and Ground Pins

Pins related to power are located here. You can use these pins to run power from your Arduino to your breadboard circuit.

Analog Pins

These pins take sensor readings in a range of values (analog), rather than just sending whether something is just on or off (digital).

Now let's connect the Arduino to your computer. We're not going to program it just yet, but it will help to see how to attach it to the computer via the USB cable.

PLUG YOUR ARDUINO INTO YOUR COMPUTER

You'll need a USB A-B cable, your computer, and an Arduino Uno. If you have a newer MacBook model, you may also need a USB-C-to-USB adapter.

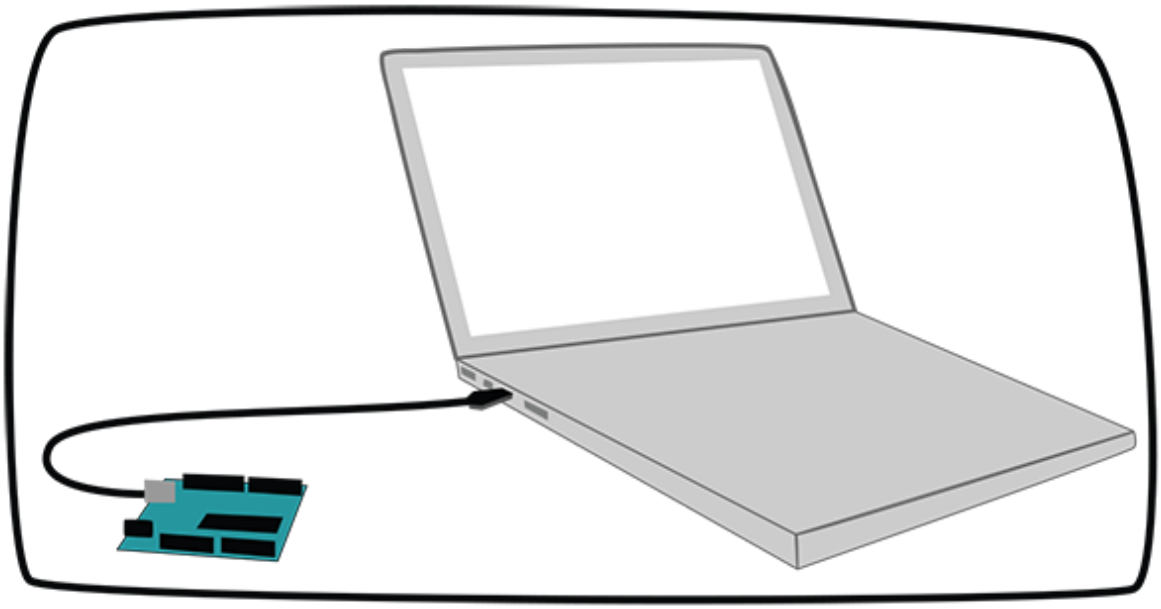


FIGURE 2-4: Connect your Arduino to your computer.

First, plug the USB cable into one of your computer's USB ports as shown in Figure 2-4. Any port that is available, as shown in Figure 2-5, should work fine.

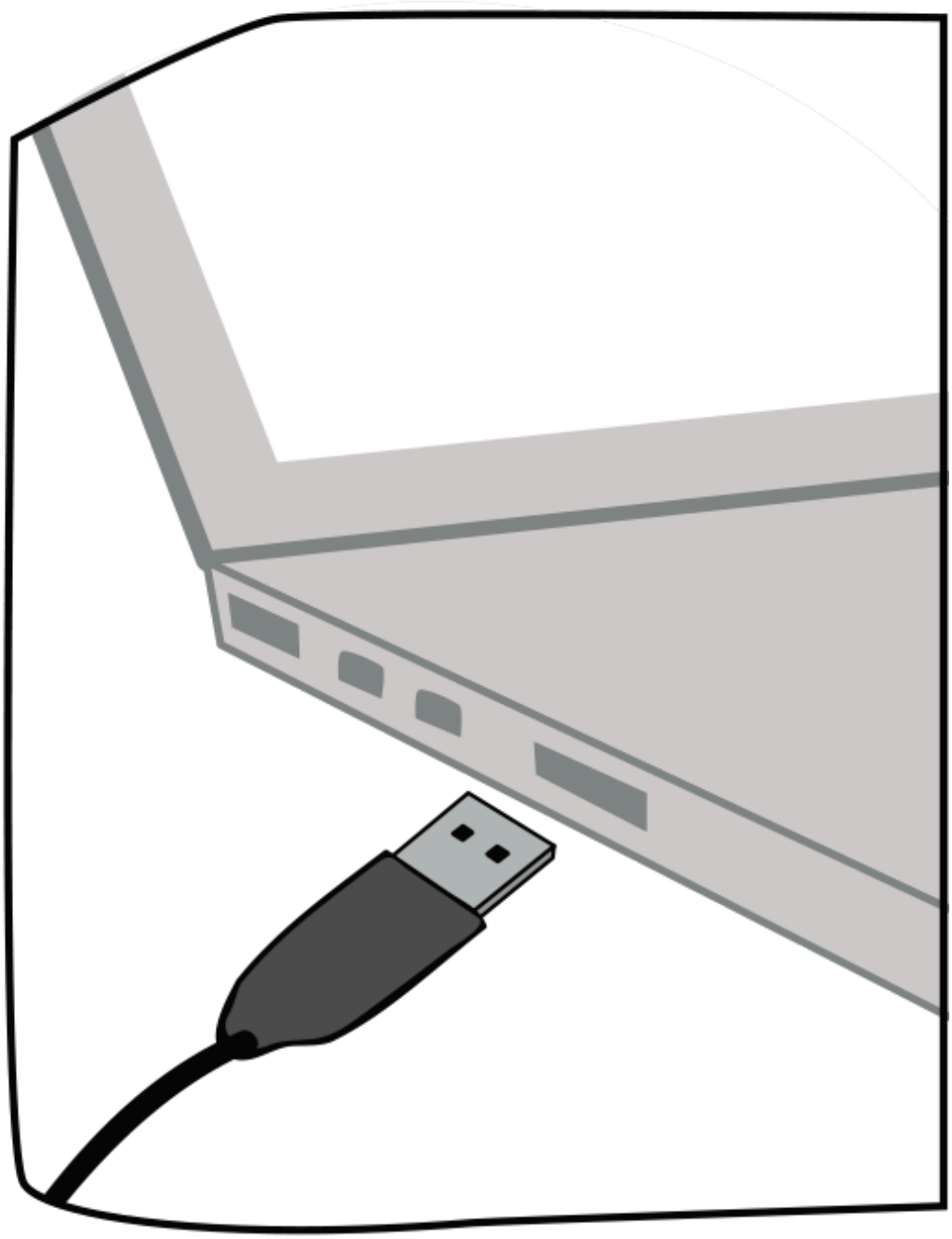


FIGURE 2-5: Close-up USB port

Now that you are attached to the computer, plug the USB cable into the USB port on the Arduino. The USB port is labeled in Figure 2-6.

FIGURE 2-6: USB port on the Arduino

You can see the top view of the USB port on the Arduino with the USB A-B cable in Figure 2-7.

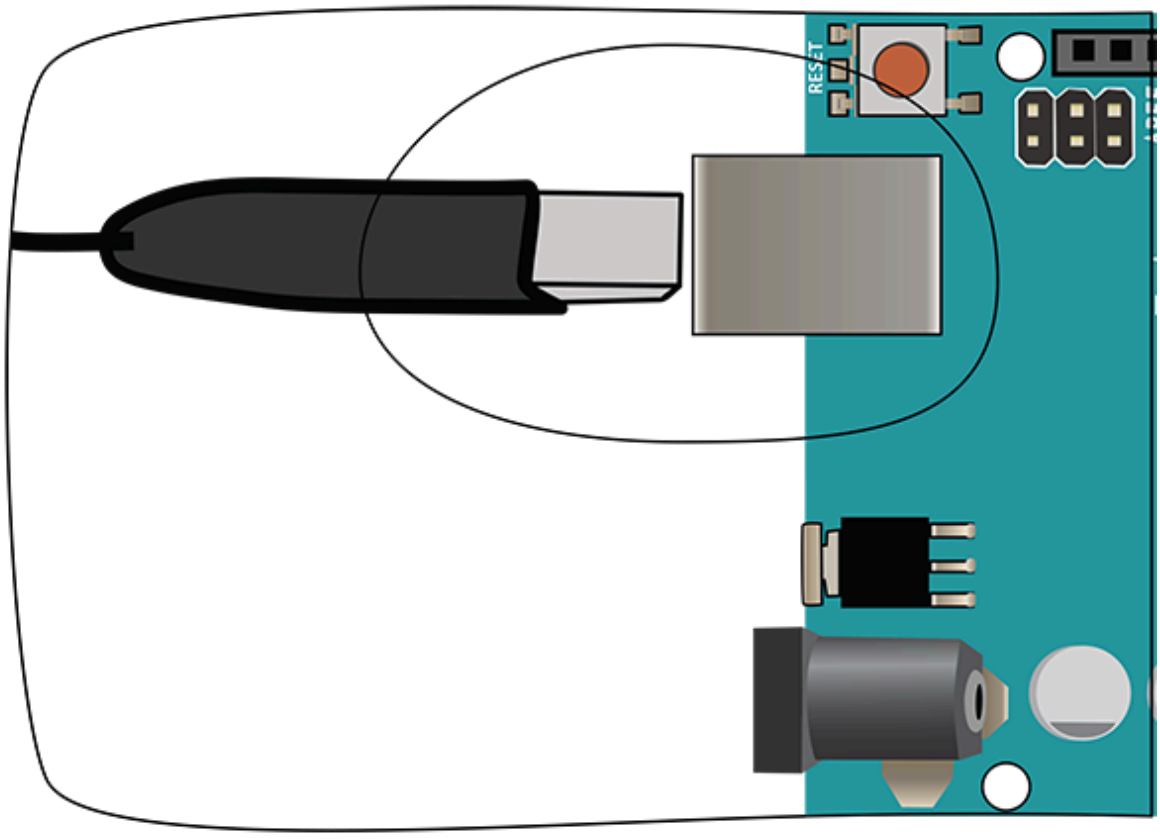


FIGURE 2-7: Top view of plugging USB cable into Arduino

PLUGGING IN THE ARDUINO

What happens when you plug in the Arduino? The power LED labeled ON should light up. And if this is the first time you've plugged it in, the LED on the Arduino near Pin 13 should blink on and off, as shown in Figure 2-8.

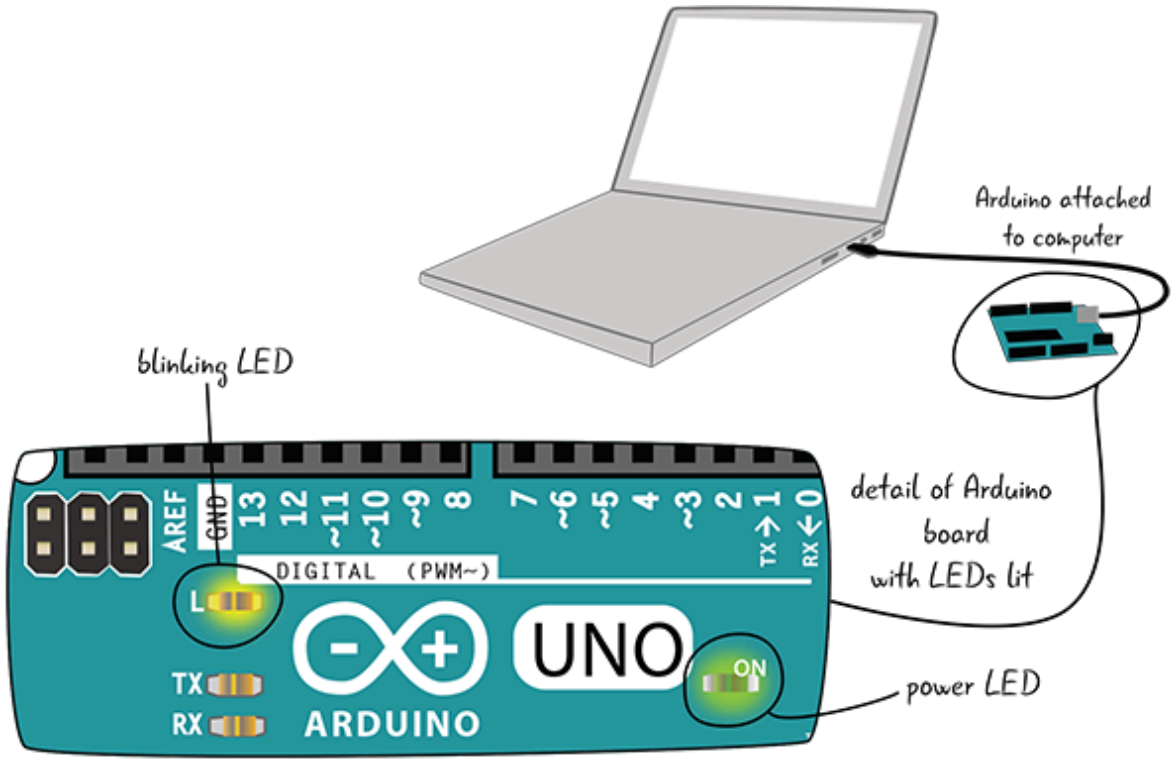


FIGURE 2-8: The LEDs turn on when the Arduino gets power from your computer.

You've Powered Up Your Arduino for the First Time!

You can always use a USB cable and a port on your computer to power the Arduino. The Arduino can also be powered by attaching it to a power supply that's plugged into a wall outlet.

> **Note**
>
> The Arduino can be powered off the USB port or the power port.

POWERING THE ARDUINO FROM A POWER SUPPLY

You will need a 9–12V DC power supply and an Arduino. The first step is to unplug the USB cable, which will completely power down the Arduino. Figure 2-9 shows the power port on the Arduino.

> **Warning**
>
> Always unplug the Arduino from a power source whenever you are making any changes!

FIGURE 2-9: Power port on the Arduino

Attach the power supply to the power port on the Arduino (Figure 2-10).

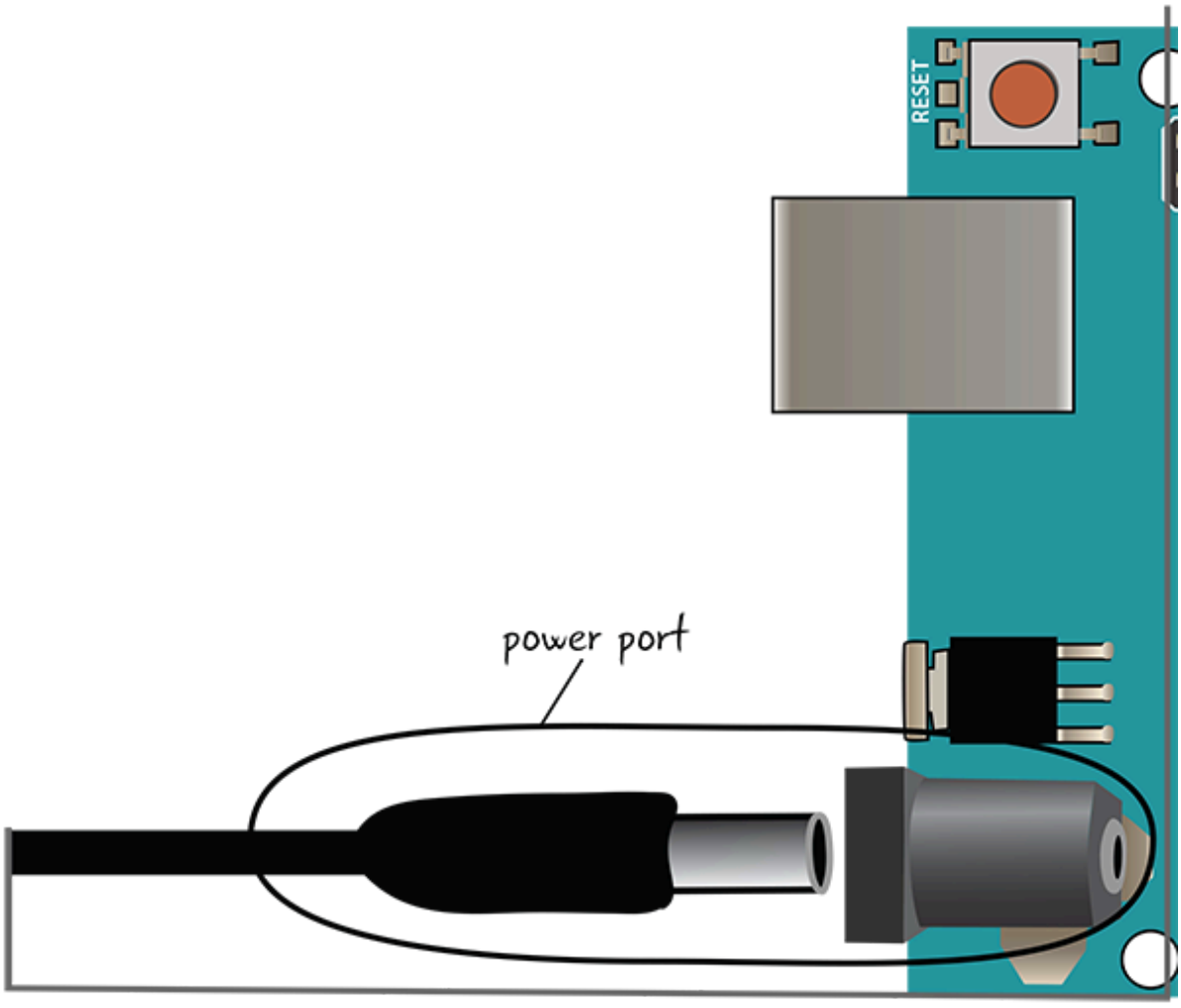


FIGURE 2-10: Top view of the power port on the Arduino

Next, plug your power supply into a surge protector, and then into a wall outlet, as shown in Figure 2-11.

What happens now? It should be just the same as when you attached the Arduino to your computer with the USB cable: the LED labeled ***ON*** indicates that the Arduino has power. And if your Arduino is straight out of the box, the LED near Pin 13 will start blinking, as seen in Figure 2-12.

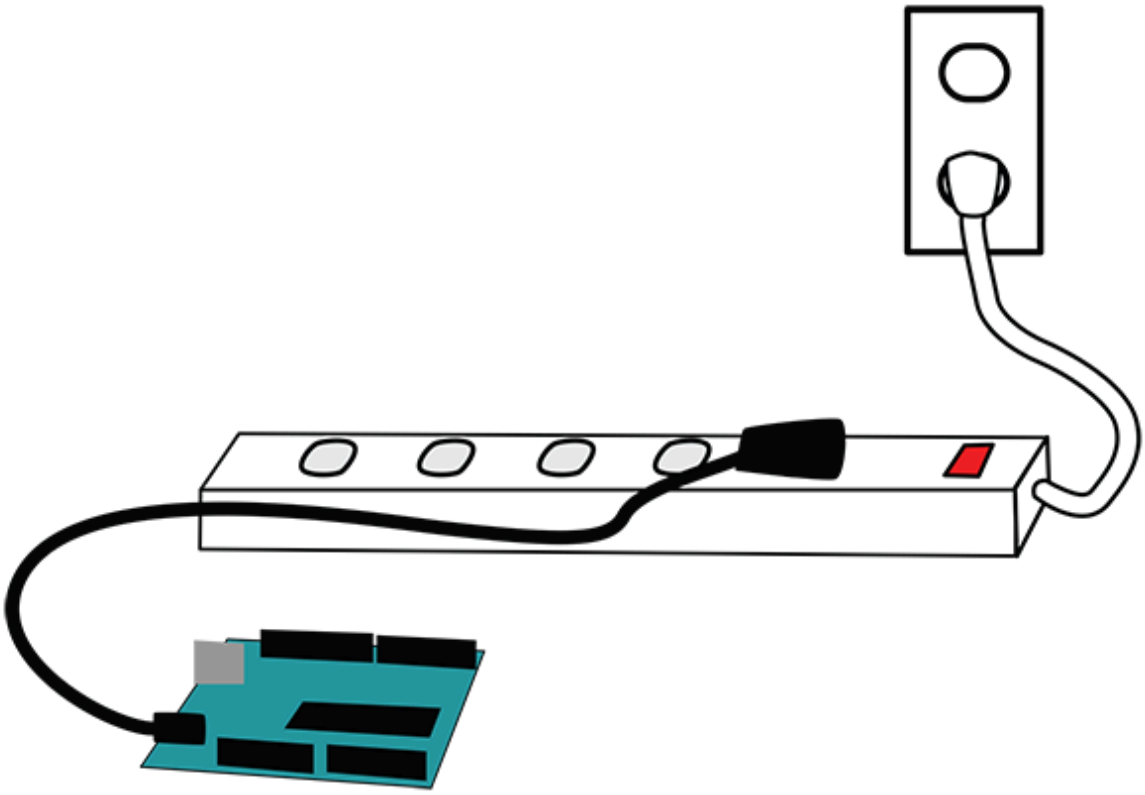


FIGURE 2-11: Plug your power supply into a surge protector.

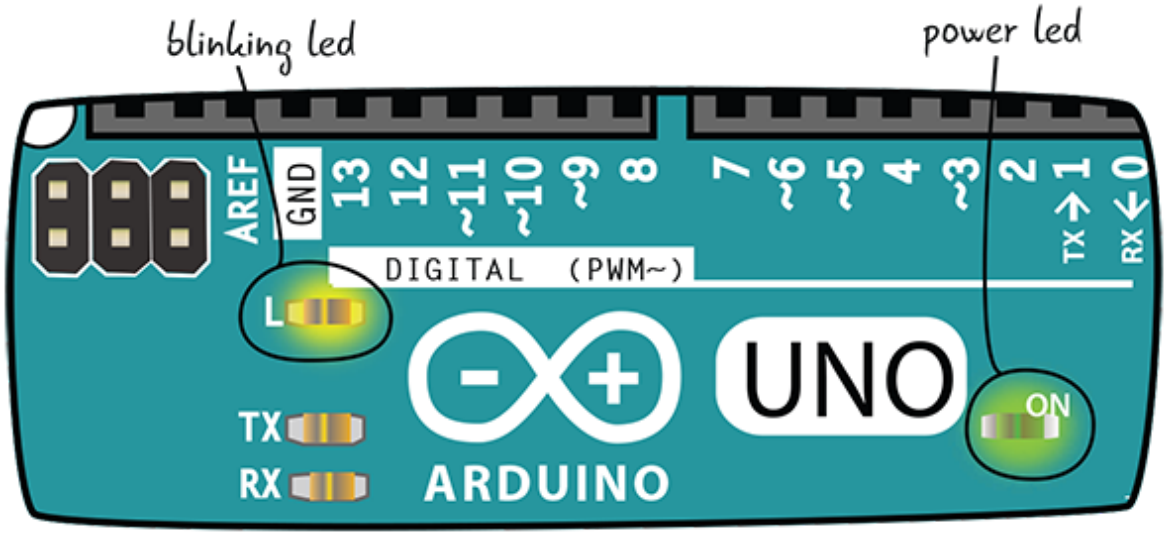


FIGURE 2-12: LEDs blinking on the Arduino

Now you know the two methods of powering an Arduino. You can switch your power source as your project evolves—you're not stuck

using one or the other.

COMPONENTS AND TOOLS

Now that you have purchased the components in the parts list (Figure 2-13), you may wish to learn more about the individual pieces. Several different types of resources are available that can help you figure out which parts to use and where to put them.

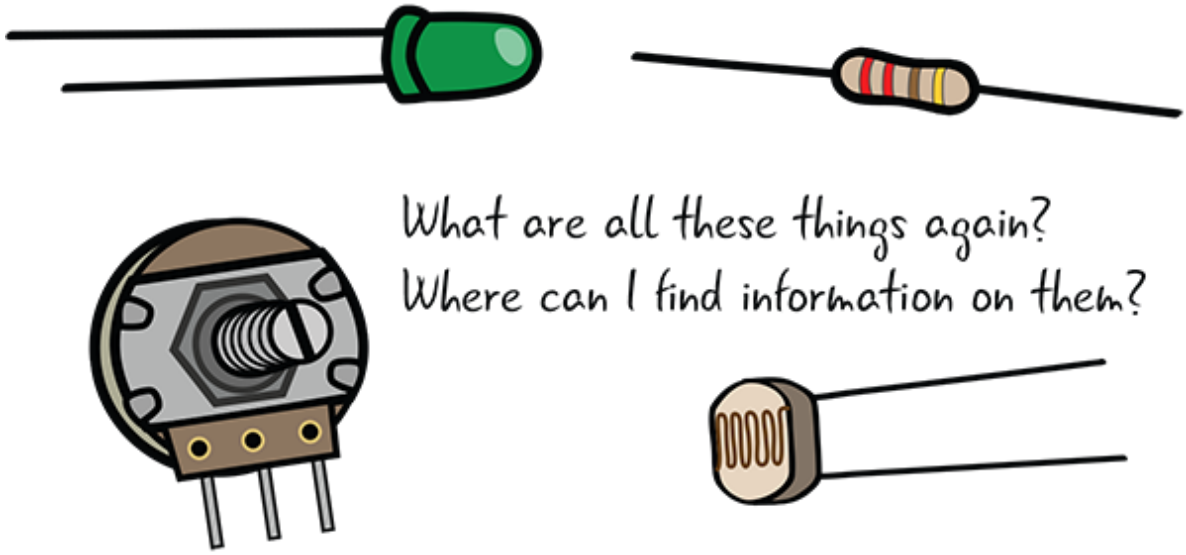


FIGURE 2-13: Where do I find information about my components?

SORTING YOUR PARTS

The best thing to do when unboxing all your parts is to separate them by type. It's nice to have all of your resistors in a separate place from your LEDs, or even to have separate places for different LED colors and for resistors of different values. Most hardware or craft supply stores sell plastic boxes that will make it easy to sort out the parts and find them when you need them later on. We recommend something that looks like the box shown in Figure 2-14.

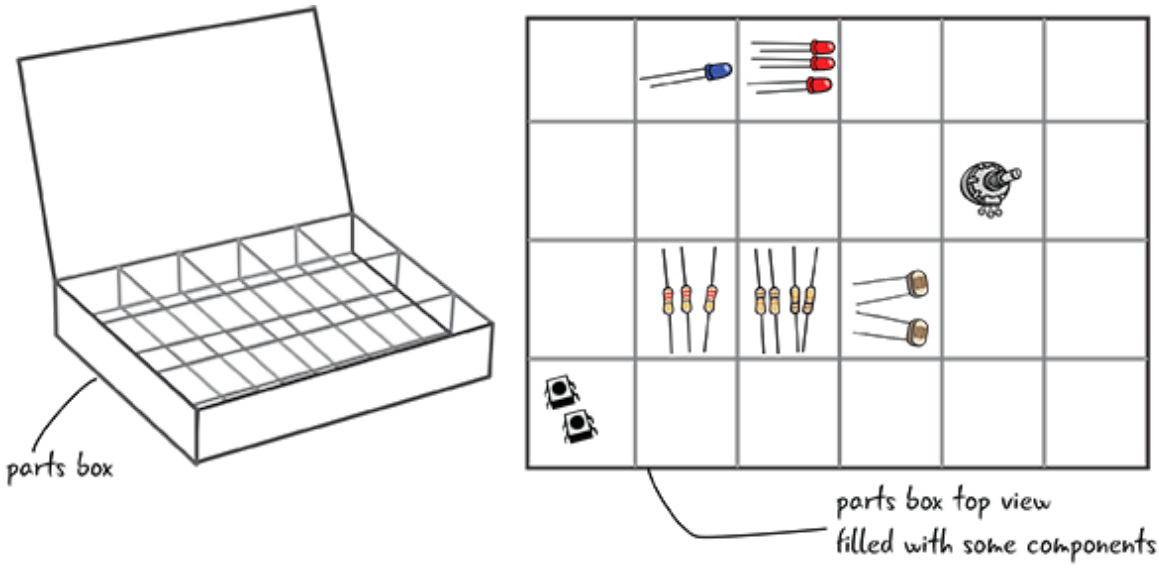


FIGURE 2-14: Sorting all of your components will also help you get familiar with them.

PART NUMBERS AND STORE GUIDES

Now that you have your parts separated out and can identify what they are, where should you look to find out information about them? The very first place to check for information about components is the components themselves. Resistors, LEDs, and most other components look different enough that you will quickly learn how to identify them. Often, components will have a part number listed on them somewhere, which can help you find a supplier's or manufacturer's website. When you order components or a kit, the store will also send along documentation or point you to a page on their website. Always check a part supplier's website first and save yourself a headache.

FINDING MORE INFO: DATA SHEETS

If you can't find the information you are looking for either on the component or on the website, the next thing to look for is your component's data sheet. You can find it by entering the part number, followed by "data sheet," in your favorite search engine online. Do not search for just the part name, since chances are there are many

different versions of your part online with different information. For example, there are a lot of different LEDs!

Electronic data sheets document the behavior, function, and limitations of electronic components. They have a tremendous amount of information, from operating temperature and behavior and suggested wiring diagrams, to material makeup and industrial application.

For example, here's how to find a data sheet online for one of your LEDs.

Find the number that identifies the LED on your invoice from the supplier you purchased your parts from. If you can't find one, use this one for red superbright LEDs: WP7113SRD.

Open a browser and type the number of your part into your favorite search engine, as well as the words "data sheet." If you use our example part number, your search terms will be "WP7113SRD data sheet."

Your search results will include data sheets about your part, often in the form of PDFs. Choose a couple of the links and click on them. Take a look at the results and make sure they approximately match the part number you searched for.

It can often be overwhelming to sift through the data sheet to find the one bit of information that you need, but data sheets come in handy, particularly when you are not sure what components you are handling. Let's start by looking at a sample sheet, as shown in Figure 2-15.

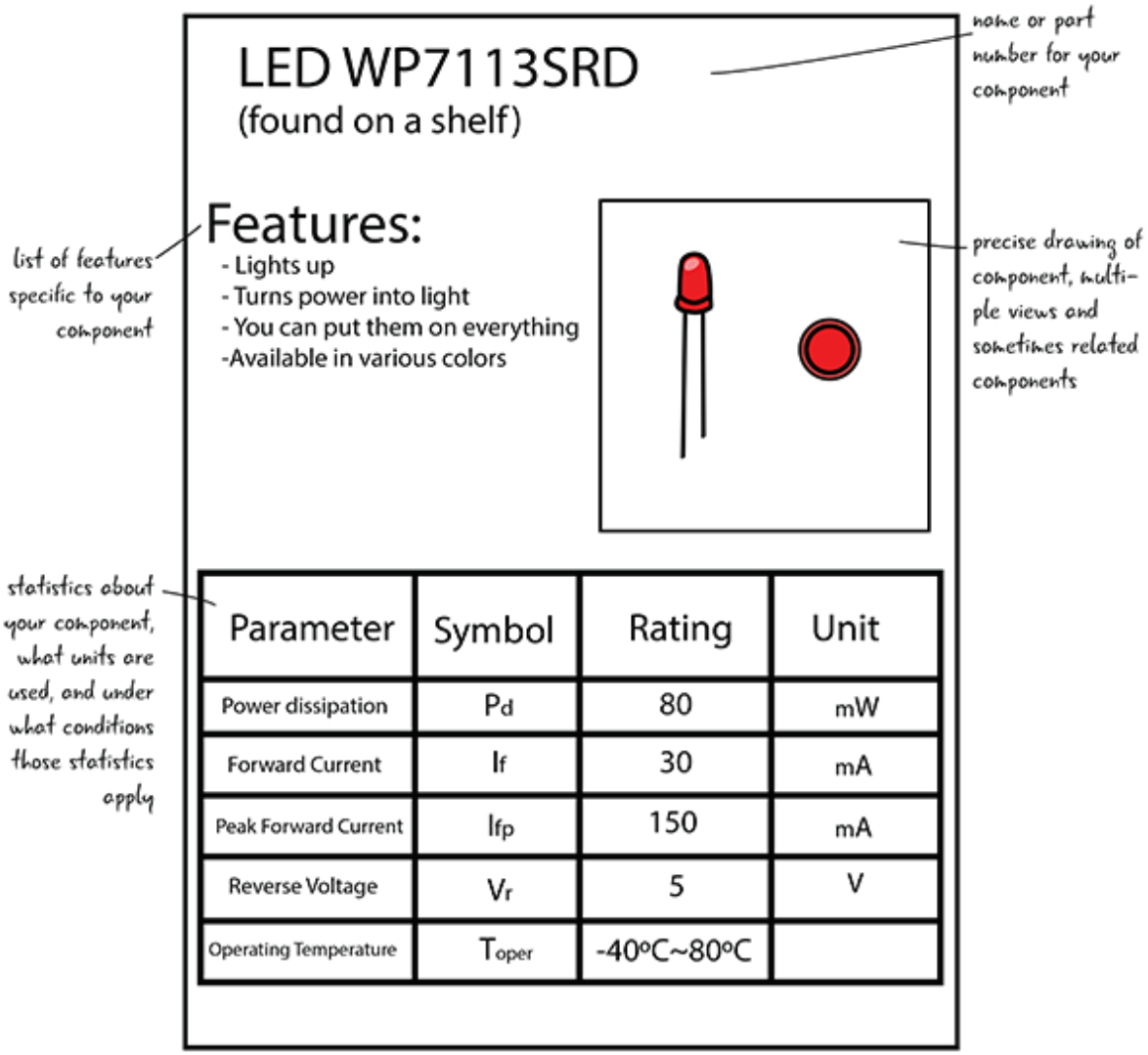


FIGURE 2-15: Data sheet for LED found on a shelf

Your data sheet contains a lot of parts, and not all of the technical information will matter for your project—but it can help you if you get stuck.

SUMMARY

You should now feel comfortable with the layout of your Arduino. You know how to power up the Arduino from the USB and the power ports. If you are ever unsure about your components, you know you can look them up online from the website where you purchased them

or search for their data sheet. In the next chapter, we are going to take a look at using a few components to build our first circuit.

3

MEET THE CIRCUIT

In the last chapter, you learned a bit about the Arduino and its parts. You were also introduced to some of the components and tools you'll be using to complete the projects in this book. In this chapter, you'll learn some of the electronics practice and theory you'll need to know to build circuits using the Arduino. We won't be using an Arduino just yet, but we'll get back to that shortly.

THE CIRCUIT: BUILDING BLOCK OF ELECTRONICS

The circuit is the basic building block for all of the electronics projects we'll be building with the Arduino.

You can build many different types of projects with an Arduino—you are limited only by your imagination. Although many different types of projects exist, all the projects in this book are built using circuits.

First, we'll look at what a circuit is; then you'll build your first circuit. We'll also look at techniques for representing electronic

circuits visually and show you how to test your circuits.

Figure 3-1 illustrates a few Arduino projects. You can see that the circuits in these projects take different forms. In the cardboard robot, you can't see the circuit, but that is what is controlling the robot.

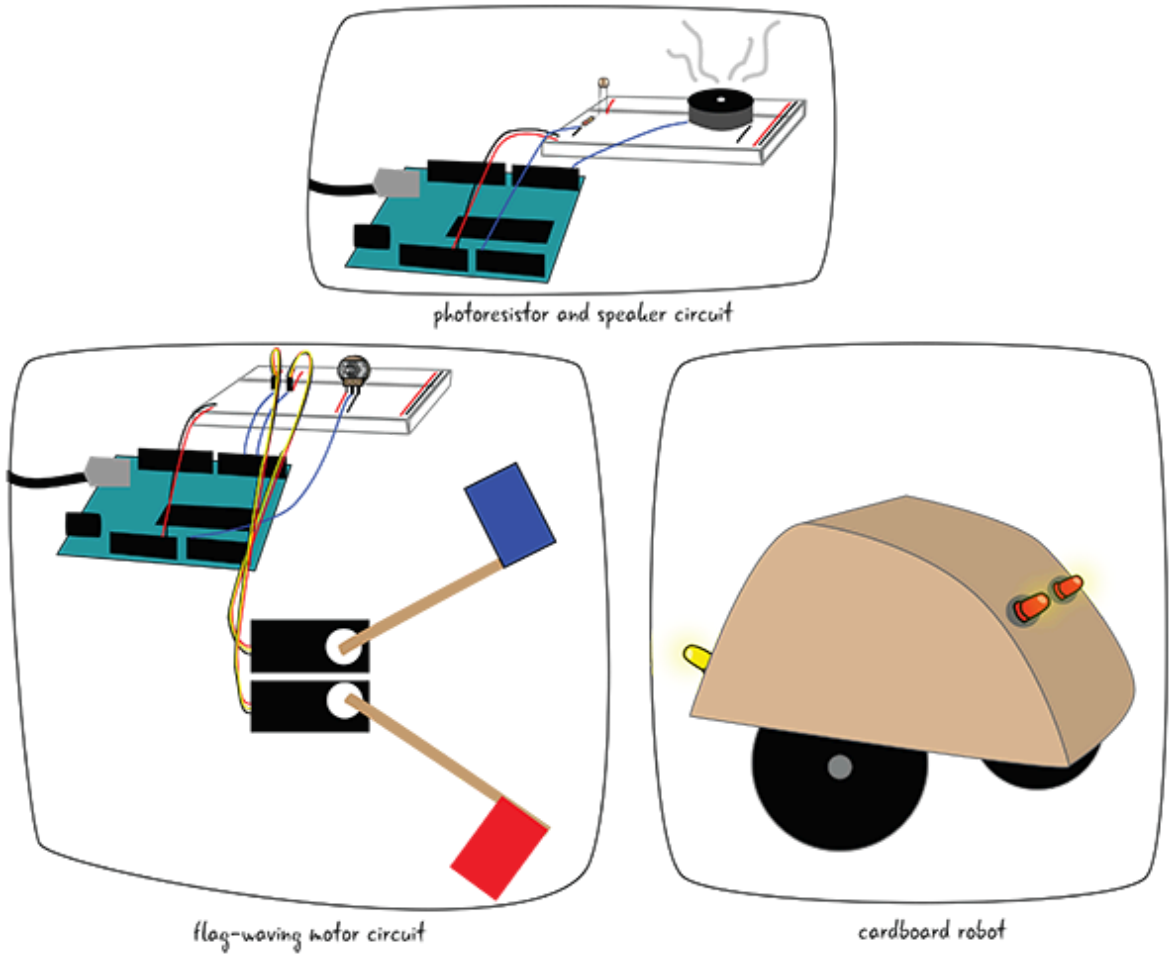


FIGURE 3-1: Some examples of projects that use the Arduino as part of a circuit

Let's look more closely at what a circuit is.

WHAT IS A CIRCUIT?

If you've ever been to a car race, you know that they refer to the track as a circuit. A circuit just means that there is a completed

closed loop, as shown in the circuits in Figure 3-2. The cars pass from the start line and end at the same place.

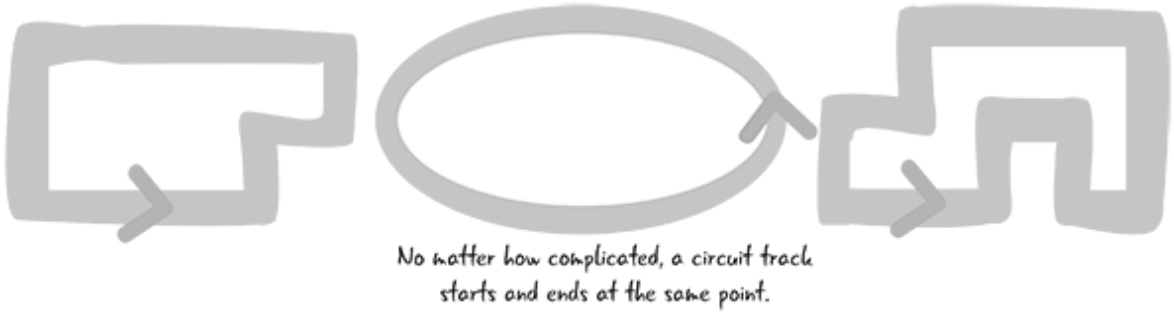


FIGURE 3-2: Circuit tracks

The same is true for electronic circuits. An electronic circuit describes a complete and closed loop. A circuit includes all of the electronic components required for a task as well as wires or another material that will let the electricity flow between the connected components, as you can see in Figure 3-3.

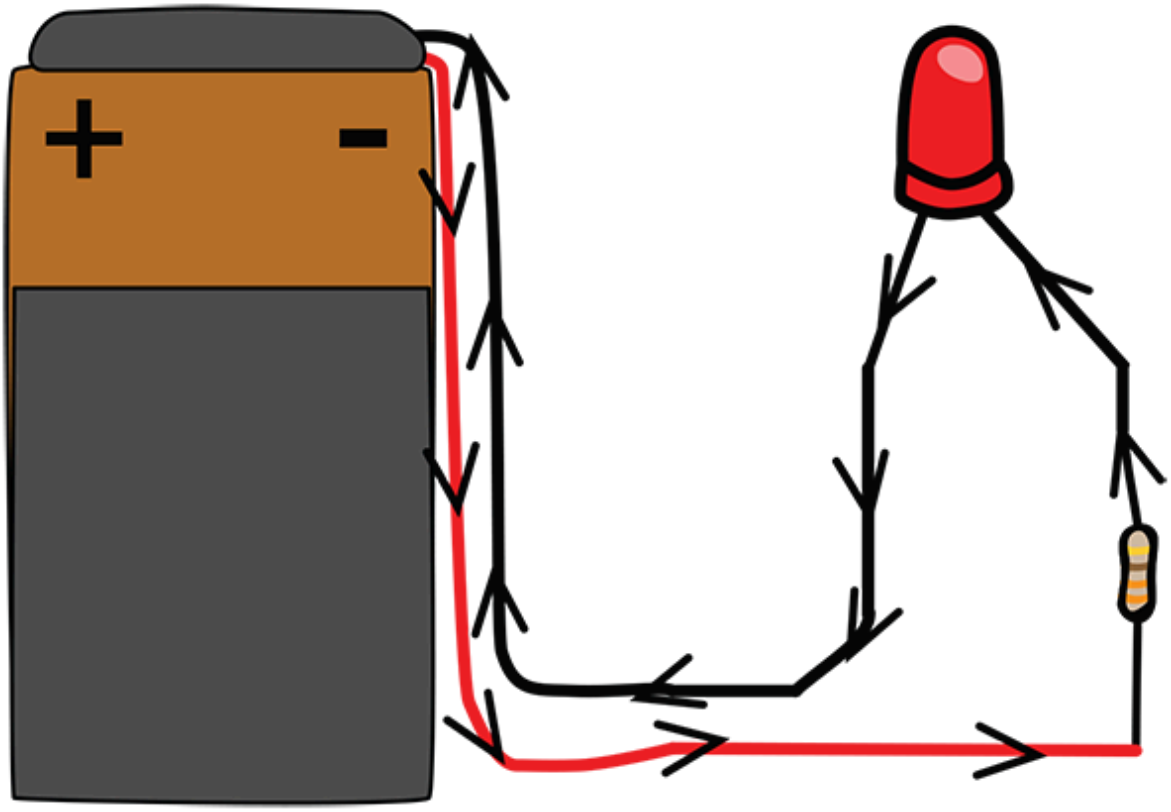


FIGURE 3-3: The flow of the circuit starts and ends with the power source.

WHY ARE WE MAKING CIRCUITS?

Think for a moment about the light switches in your home as a model. To turn a switch on or off, you must be in physical contact with the switch. In our projects, the Arduino will control the behavior of the electronic components. Our electronic components will be arranged in a circuit, and the Arduino must be part of that circuit in order for it to control the behavior.

Circuits allow the Arduino to connect to the electrical components, turning off and on a variety of objects (speakers, LEDs, motors, etc.) or taking information from the outside world ("How hot is it?"; "Is the switch on?"; etc.). As long as we figure out how to have the Arduino connect to the object, we can control it with electricity and, later, programming.

WHAT MAKES UP A CIRCUIT?

There are two main parts that make up a circuit: *conductive lines* and *components*.

Conductive Lines

Although most of the focus for a circuit is placed on the components, you cannot have a circuit without some sort of connection between the components. Our computers and electronic devices contain printed circuit boards (PCBs). PCBs, which do not conduct electricity, are composed of base layers of material onto which fine lines of conductive material have been applied, as seen in Figure 3-4. The conductive lines connect components that are soldered to the PCB. If you look at a PCB, you'll notice the shiny silver lines running between the components, connecting them. These lines are like wires stuck to a flat surface.

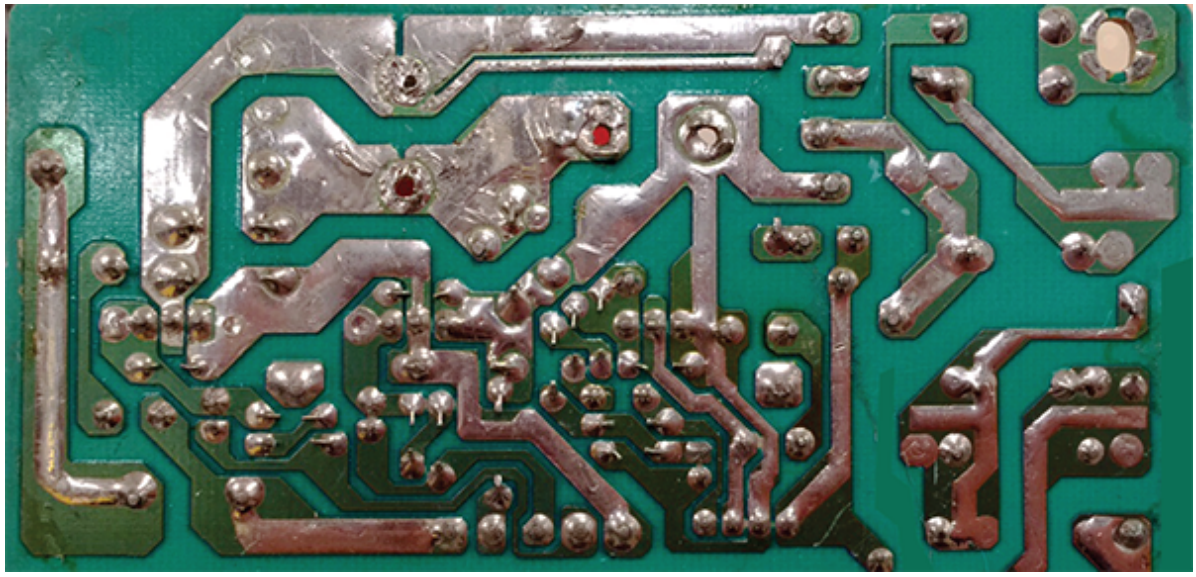


FIGURE 3-4: Detail of printed circuit board

Components

Components are the other requirement for a complete circuit. We looked at a whole list of components to buy in Chapter 1,

"Introduction to Arduino." The components form the locations that need to be connected within a circuit (Figure 3-5).

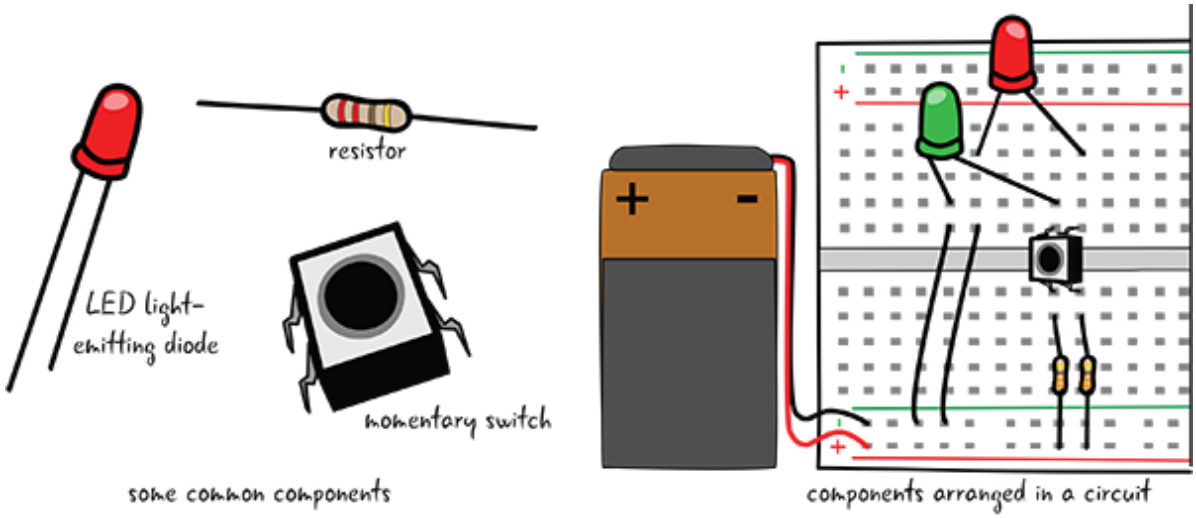


FIGURE 3-5: Circuits are made of components.

In Figure 3-6, you can see that the leads of the components are acting as conductive lines.

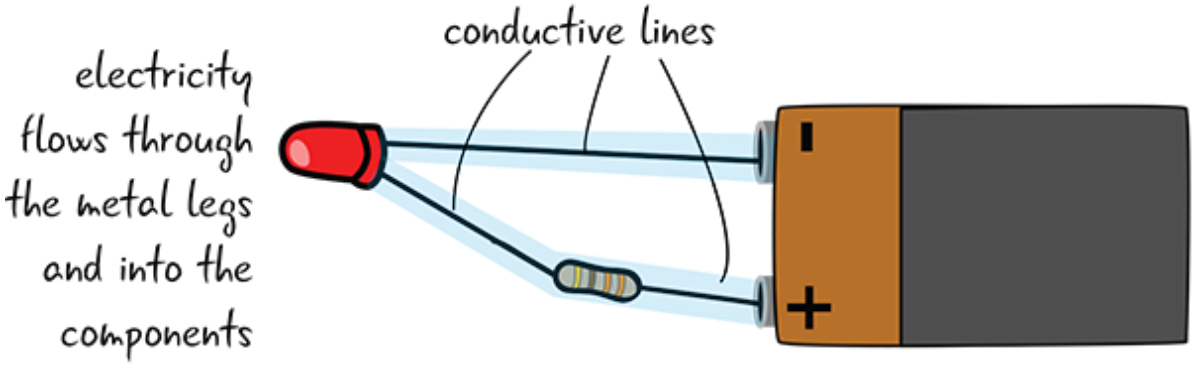


FIGURE 3-6: Electricity flows through conductive lines.

WHERE DO WE BEGIN?

The first circuit we're going to build together is an LED bulb flashlight powered by a battery. This circuit is a great beginner project because the light turning on confirms visually that the circuit is working. The

flashlight circuit also demonstrates the basic techniques of circuit building you'll need throughout all the projects in this book.

Figure 3-7 is a drawing of the circuit when completed, with the parts annotated. We'll explain what the parts do in detail, partially in this chapter and in forthcoming chapters as well. For now, know that this circuit will be built from an LED, a resistor, a jumper, a 9V battery, and a battery cap arranged on a breadboard, components you met in Chapter 1.

There are many different ways of representing or drawing circuits to convey the necessary information. In Figure 3-7, we have made an approximation of what the circuit will look like when you build it. This isn't always the clearest way to see what is happening—some circuits have many parts that are connected in complex ways. Schematics are a great way to make a drawing of a circuit that has simplified parts and show how they are connected. Let's take a closer look at how schematics work.

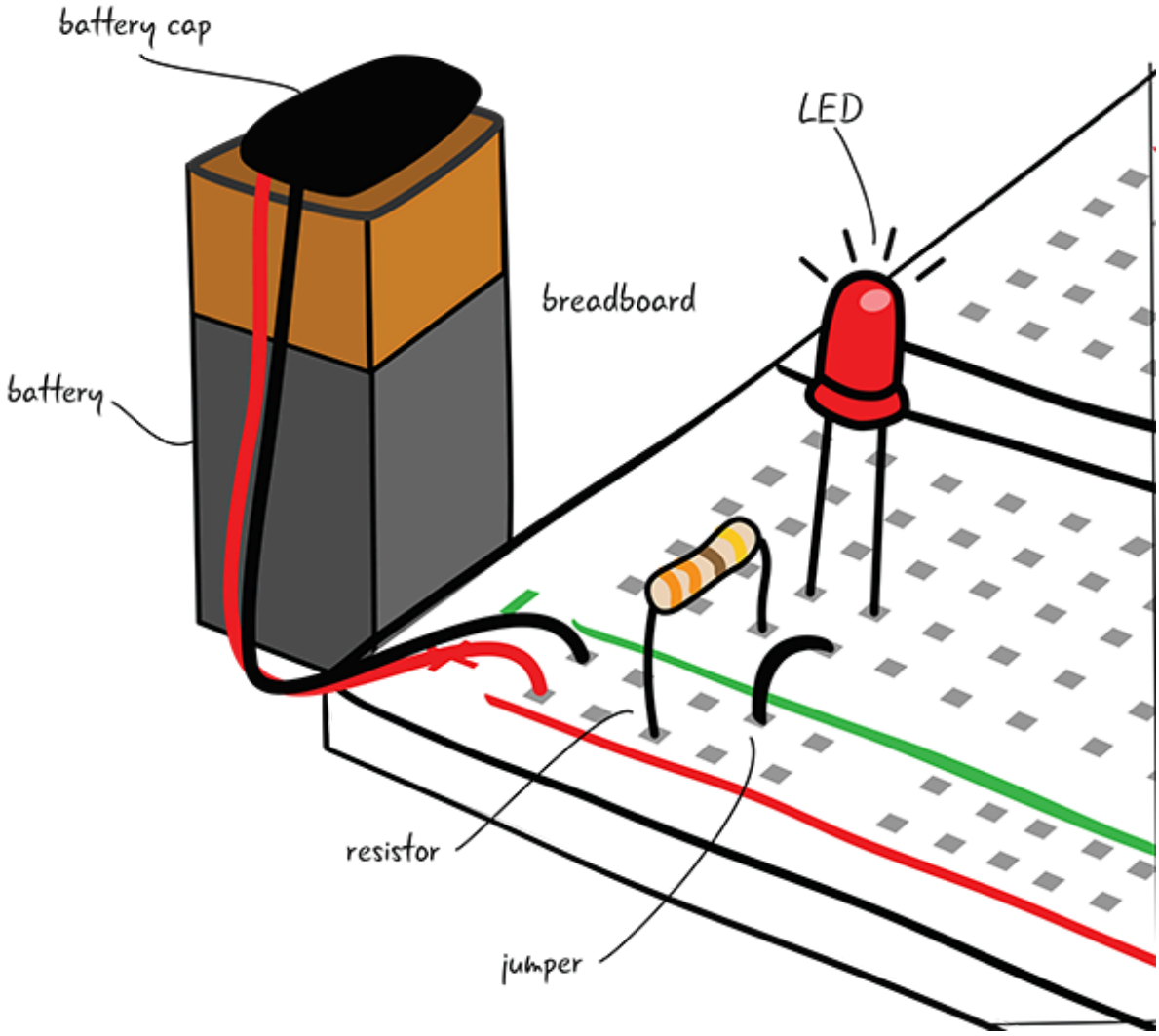


FIGURE 3-7: The circuit we'll build

THE SCHEMATIC

A schematic is a diagram of the *relationships of the electronic components in a circuit*. In a schematic, you see the components that are part of the circuit and how they are attached to each other. Let's start by looking at a simple schematic that represents our basic circuit. We'll get into the details about what each symbol means in the schematic soon, but for now let's just take a quick look. Figure 3-

8 compares a schematic of the circuit we are about to build to a drawing of the circuit.

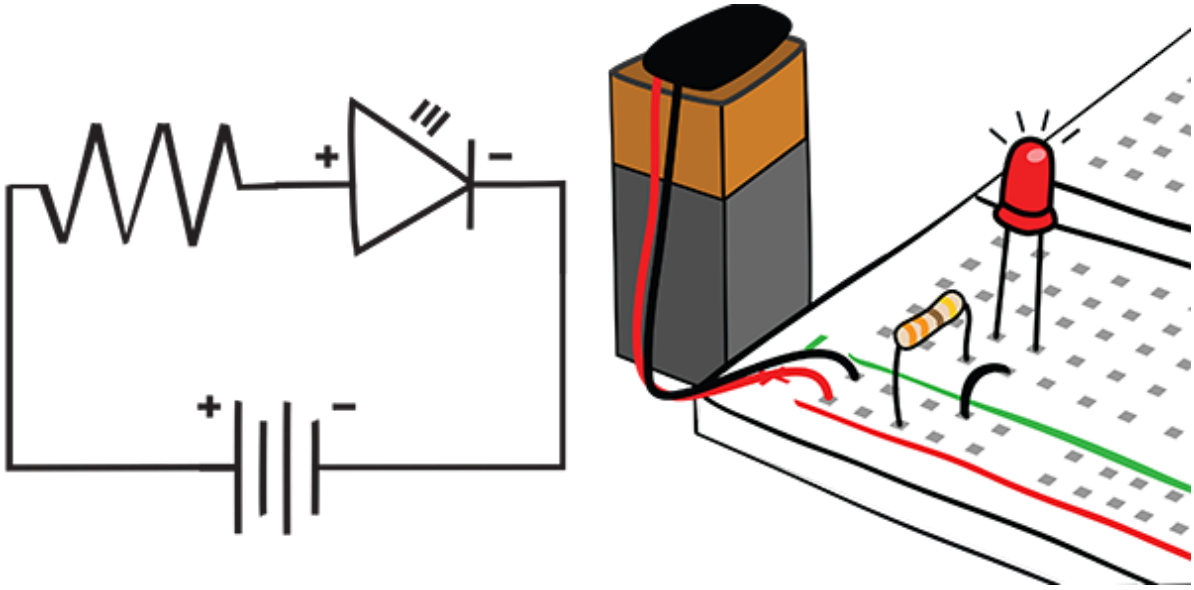


FIGURE 3-8: Schematic of the circuit with a drawing of the circuit

WHY IS IT IMPORTANT TO LEARN HOW TO READ A SCHEMATIC?

Most electronic projects and components are represented by schematics, not necessarily by drawings or photographs. As your electronic skills advance and you want to build your own projects outside of this book, you'll need to be able to read and draw schematics in order to research your projects, as well as describe and build them.

We're starting with simple schematics—we'll build up to more complex representations as we build more complex projects in the book. As you look at schematics online or in other documentation, you may notice that there are sometimes variations in how the symbols are drawn or arranged. Don't worry if all the schematic symbols don't look exactly alike, as shown in Figure 3-9.

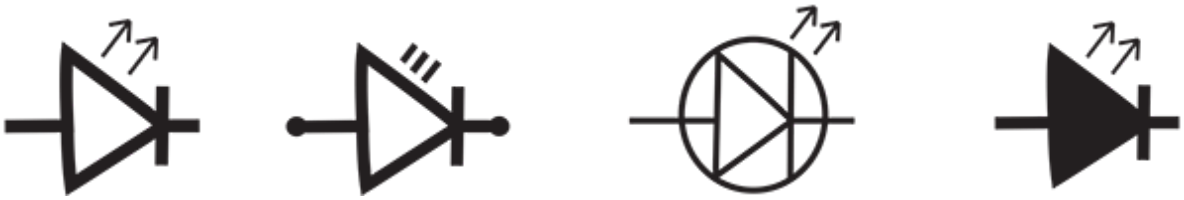


FIGURE 3-9: Schematic symbols for LEDs

DIAGRAM OF YOUR CIRCUIT: THE SCHEMATIC

You've learned that a schematic is the standard way to represent the electrical relationships in a circuit. All commonly used electronic components have a symbol to represent them within electronic schematic diagrams in order to make it clear what is attached within the circuit. Figure 3-10 shows a basic circuit of one LED, a resistor, and a battery. The LED has an orientation, a positive lead (anode) and a negative lead (cathode), as mentioned in Chapter 1.

Schematics are primarily concerned with diagramming how the components are connected in the circuit, and will sacrifice clarity in how the components are set up physically to demonstrate better how they are connected electronically.

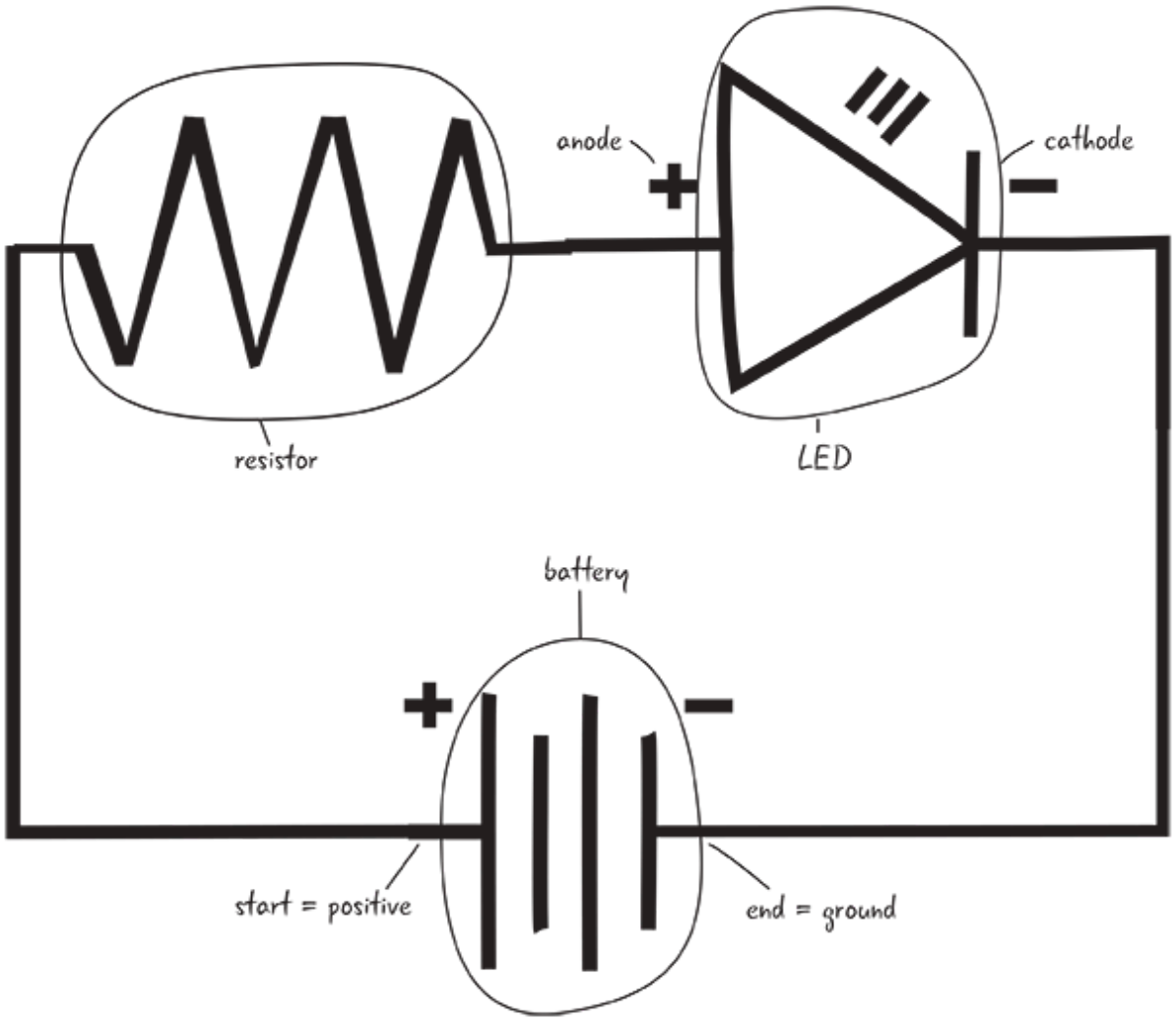
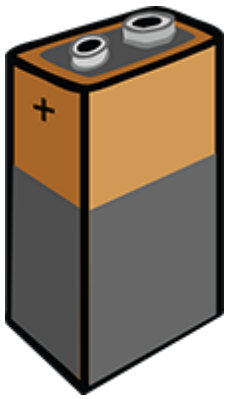
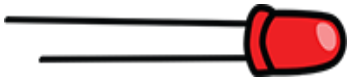


FIGURE 3-10: Annotated schematic for the circuit

Table 3-1 shows the symbols for the components that are in our first circuit. The Wikipedia page on electronic symbols is a good place to get an overview of many of the symbols used in schematics: en.wikipedia.org/?title=Electronic_symbol.

TABLE 3-1: Components with their schematic symbols

COMPONENT	DESCRIPTION	SCHEMATIC SYMBOL
	Battery	
	LED (light-emitting diode)	
	Resistor	

There are also a few other ways that the symbols from a power source can be drawn, as you can see in Figure 3-11. We'll cover the concepts of power and ground later on in the chapter, but recognizing these symbols will help you understand what is going on in our circuit.

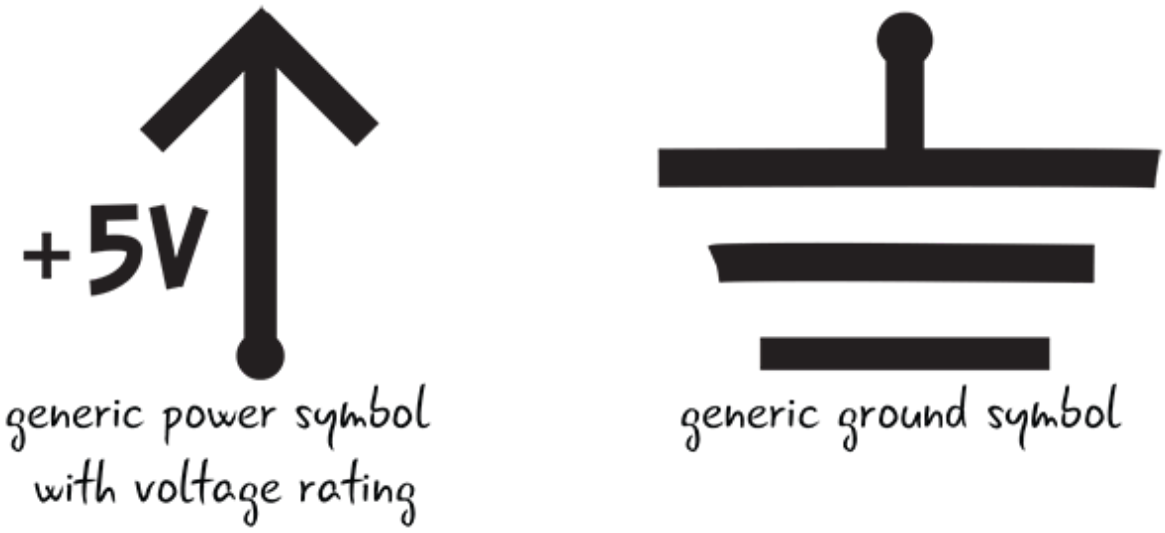


FIGURE 3-11: Schematic symbols for power and ground

DRAWING A SCHEMATIC

You've seen an example of a schematic, as well as the symbols that are used in the schematic for our first circuit. How do you connect the symbols to draw a schematic?

We'll start with the symbol for a resistor in Figure 3-12. Remember that the resistor does not have a positive-negative orientation, so it does not matter which end is which.

FIGURE 3-12: Schematic symbol for a resistor

We'll next draw the symbol for the LED and connect it to the resistor with a solid line. Why is the line solid? Remember that we are representing the physical connection between the components in the circuit, just like the conductive silver lines on the PCB.

The positive end, or anode, connects to the resistor as it will in the circuit when we build it, as seen in Figure 3-13. When we attach the battery, the power will flow through the resistor to the positive end of the LED.

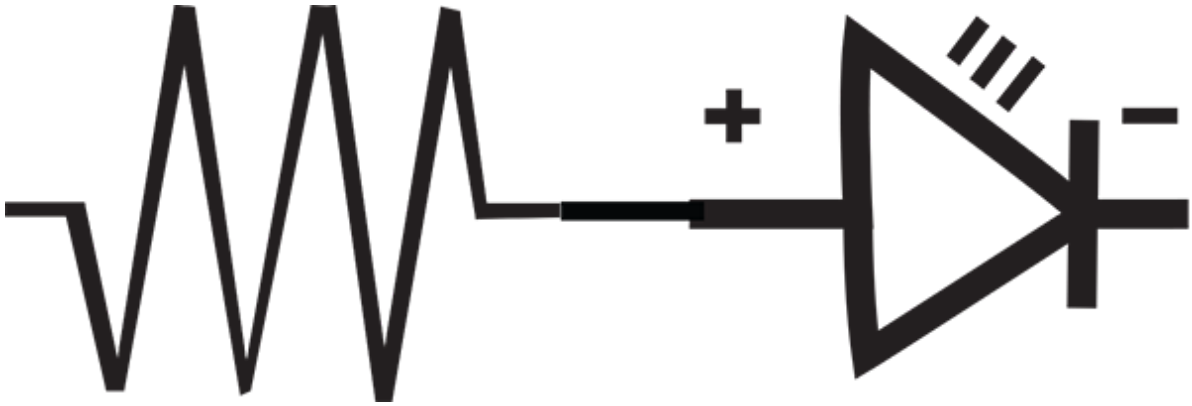


FIGURE 3-13: Resistor connected to anode of LED

Now we add the symbol for the battery and connect it to the symbols for LED and resistor, as shown in Figure 3-14. The negative end of the LED, or cathode, connects to the negative end of the battery.

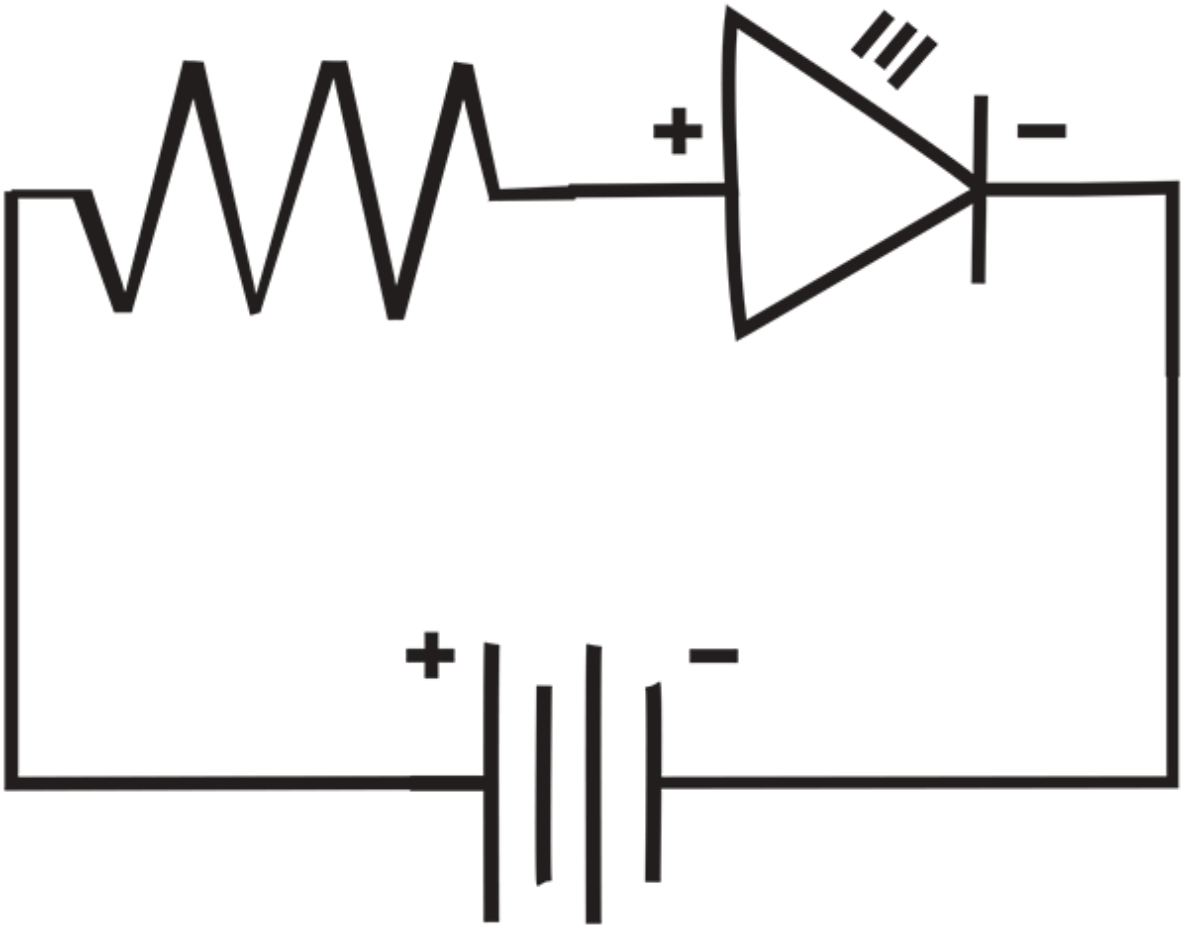


FIGURE 3-14: Schematic for the circuit

We can see in this schematic that one end of the resistor is attached to power, or the plus sign on the battery. The other end of the resistor is attached to the positive end of the LED. The negative end of the LED is attached to ground, or the minus sign. Our schematic represents the complete loop of our circuit.

USING A BREADBOARD

How do we attach the components to build a circuit? If you take a look at Figure 3-15, you can see there is a *breadboard* beneath all the components.

Why do we use a breadboard? The breadboard allows us to connect all our components. We could never hold all the pieces together with our fingers, and we don’t want to permanently attach them to each other initially. We know that a circuit is a loop and that the components must be connected. The breadboard allows us to connect our components to each other rapidly and gives us the flexibility to easily adjust our circuits. Using a breadboard allows us to rapidly prototype our projects.

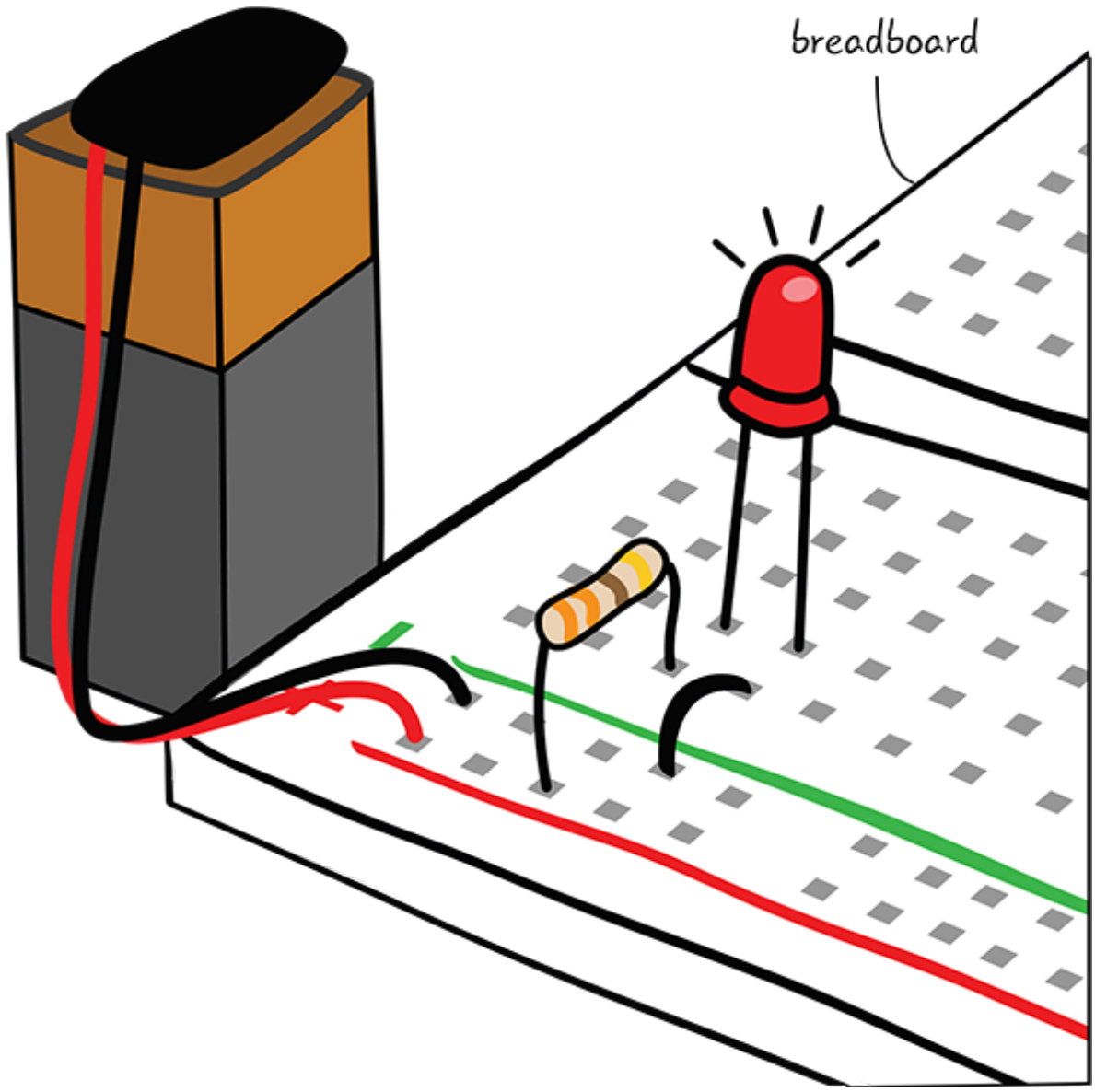


FIGURE 3-15: The circuit we'll be building, with the breadboard marked

Note

Using a breadboard allows us to attach components to each other quickly and make adjustments to our circuit.

BREADBOARD BASICS

You've seen pictures of a breadboard and circuits assembled on a breadboard. You also know that using a breadboard allows you to quickly prototype circuits and test them out. How is a breadboard constructed? Let's look at an "x-ray" view of a breadboard.

Warning

Don't actually remove the backing—doing so could ruin your breadboard.

A breadboard has strips of metal encased in plastic with a grid of holes on the top. The holes, called *tie points*, are placed at regular intervals and arranged in rows and columns.

In Figure 3-16, you can see the metal strips arranged over rows and columns of tie points. All of the tie points that are connected to one of the metal strips are connected to each other.

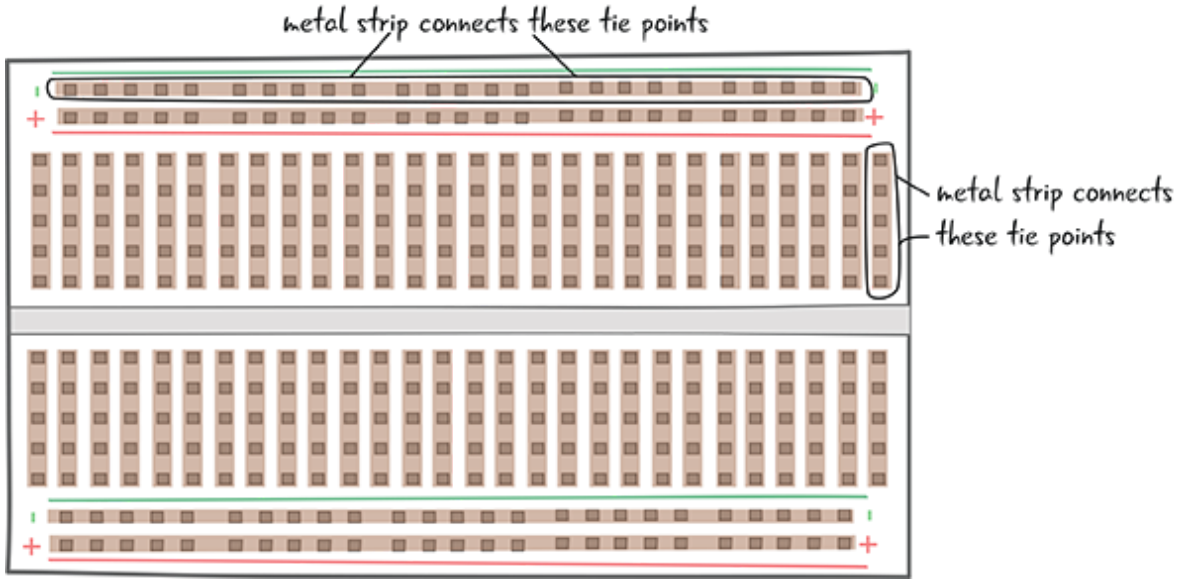


FIGURE 3-16: An "x-ray" view of a breadboard

The rows and the columns are arranged in patterns to make it easy to build circuits with standard electronic components.

The long columns on the far left and right of the board shown in Figure 3-17 are by convention attached to *power* and *ground*, and they are called *power and ground buses*. There is a plus (+) sign or a minus (–) sign at the top of each column. They will be attached to the plus and minus signs on the battery. There is often a red line close to the power bus, and a green, blue, or black line next to the ground bus. Some breadboards, particularly smaller ones, do not have these power buses.

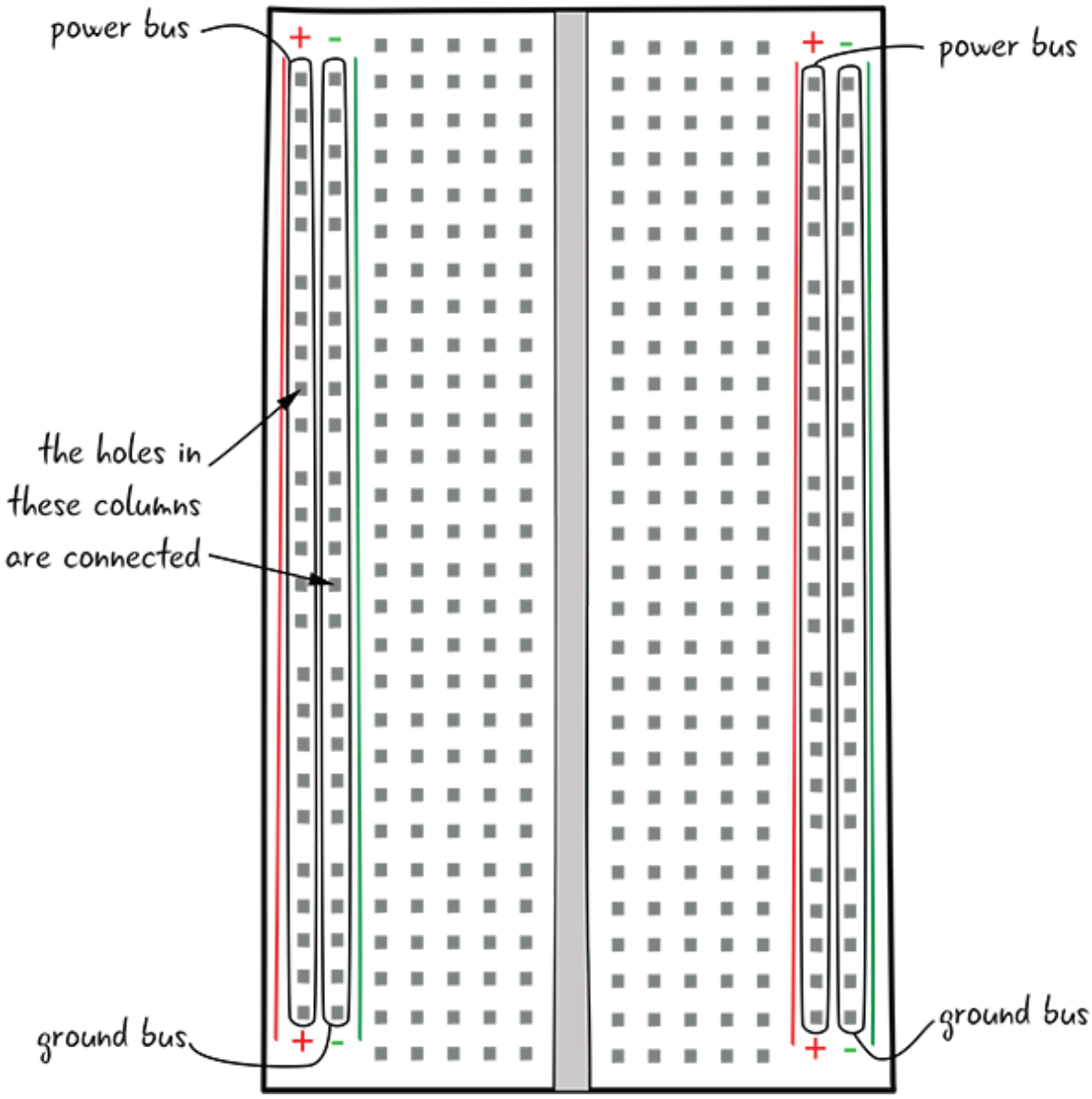


FIGURE 3-17: Power and ground buses in a breadboard

We'll explain more about power and ground later. For now, you just need to know that we'll connect a battery to the buses on one side of the board, and the left and right side buses are not connected. Left or right, it doesn't matter to which side of the breadboard you attach power and ground, though we'll connect the battery to the left side of the board. It's a good idea to be consistent in how you set up your breadboard.

MAKING CONNECTIONS

Generally a gap, known as a trench, exists down the middle; it is the same width as some components, to make it easy to plug them into the circuit. The tie points in each row on either side of the trench are connected, allowing you to make connections between components when you place them on the board. Figure 3-18 shows that they *do not* connect across the trench; each row of tie points on either side of the trench is a discrete row.

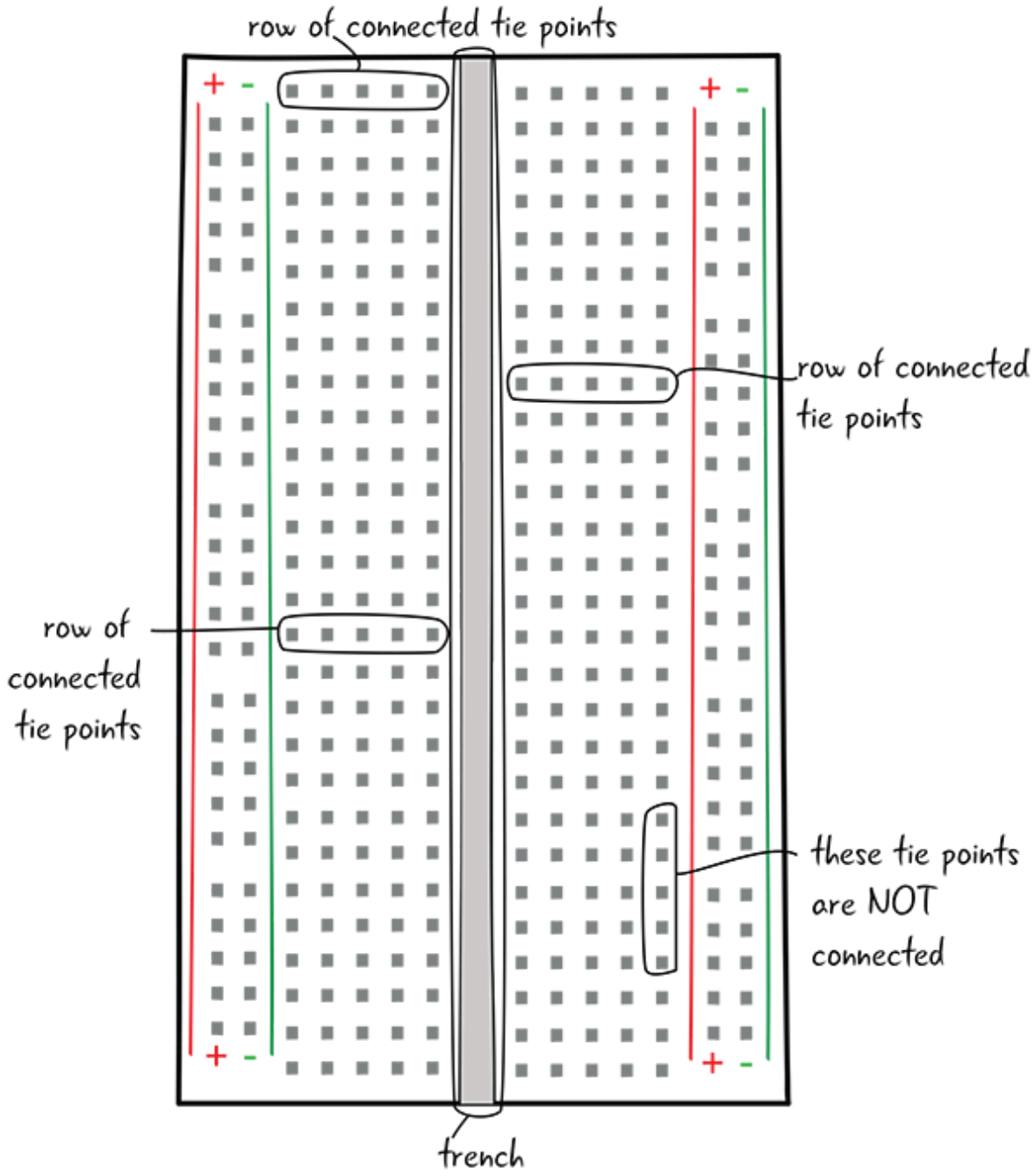


FIGURE 3-18: Row of tie points in the breadboard

Note

The rows in a breadboard do not connect across the trench.

Components can be connected to each other by putting them in the same row of tie points, as shown in Figure 3-19.

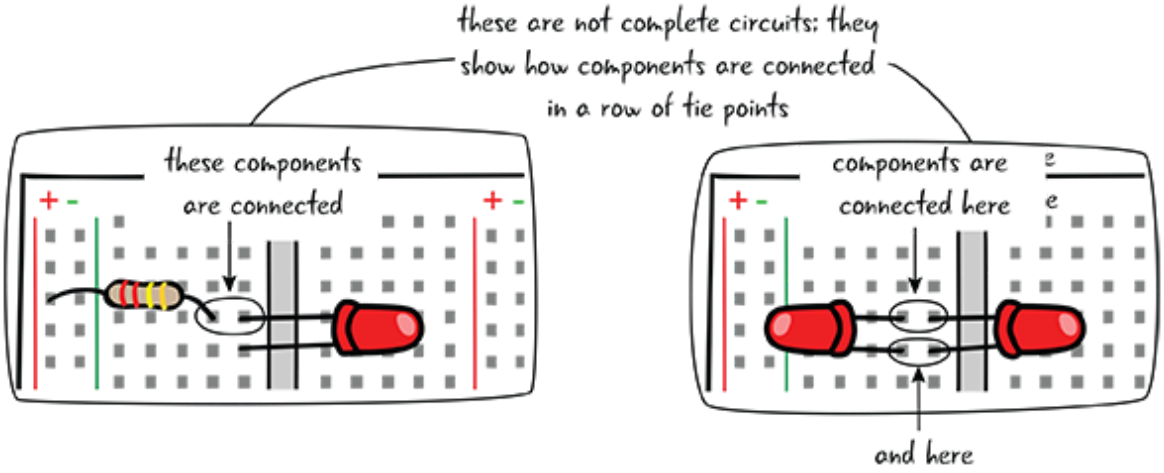


FIGURE 3-19: Connected components in breadboard

QUESTIONS?

Q: Do I need a new breadboard for each circuit I build?

A: The great thing about breadboards is that it is very easy to change out the parts of a circuit or make a new one entirely. You could make all of the circuits in the book by just reusing one breadboard. If you want to have more than one circuit set up at once, it is helpful to have an additional breadboard.

BUILDING A CIRCUIT

We're going to build our first circuit! You'll need these parts and tools:

Breadboard

9V battery

Battery cap

1 LED

330-ohm resistor (bands colored orange, orange, brown, gold)

Jumper wires

Needle-nose pliers

Get all of your parts together to start building the circuit in Figure 3-20.

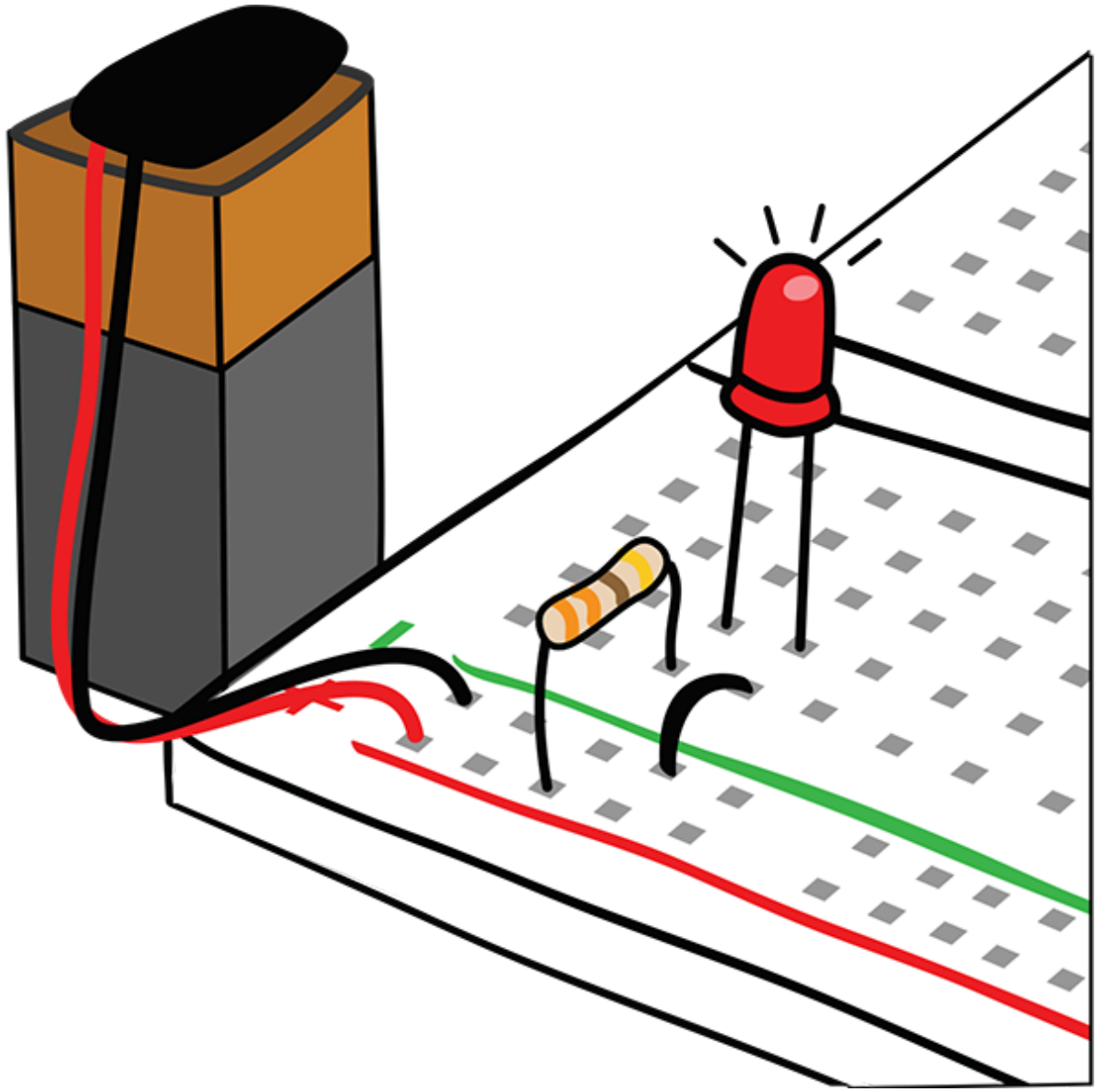


FIGURE 3-20: The circuit

STEP-BY-STEP CIRCUIT INSTRUCTIONS

We are going to walk you through the steps of making the basic circuit we've shown you throughout the chapter. You may not understand exactly how all the parts in the circuit work together yet. Don't worry about this—we'll explain more about electricity in a circuit and about each component as we move forward. For now, just follow the steps.

The first parts you'll need are the breadboard and the 330-ohm resistor. You'll learn more about resistors later, but right now you just need one resistor that has four bands with the colors orange, orange, brown, and gold.

Pick one corner of the breadboard—we are starting with the upper-left corner. (It doesn't make a difference if you pick the right- or left-hand buses, but it's preferable to be consistent.) First put one end of the 330-ohm resistor (with bands colored orange, orange, brown, and gold) into the power bus (marked with the + sign) of your breadboard and the other end into a row of your board. You'll have to bend the leads a little so you can get them into the board.

Resistors don't have a forward or backward direction in a circuit, so it doesn't matter what the orientation is. Each lead or leg is the same. Figure 3-21 shows how the resistor is attached.

Tip

The components should feel like they are pressed in place. Sometimes it's hard to get the components all the way into the board. Just be patient. Some people find it easier to use needle-nose pliers to stick components into a board, whereas others just use their hands. See what's easier for you.

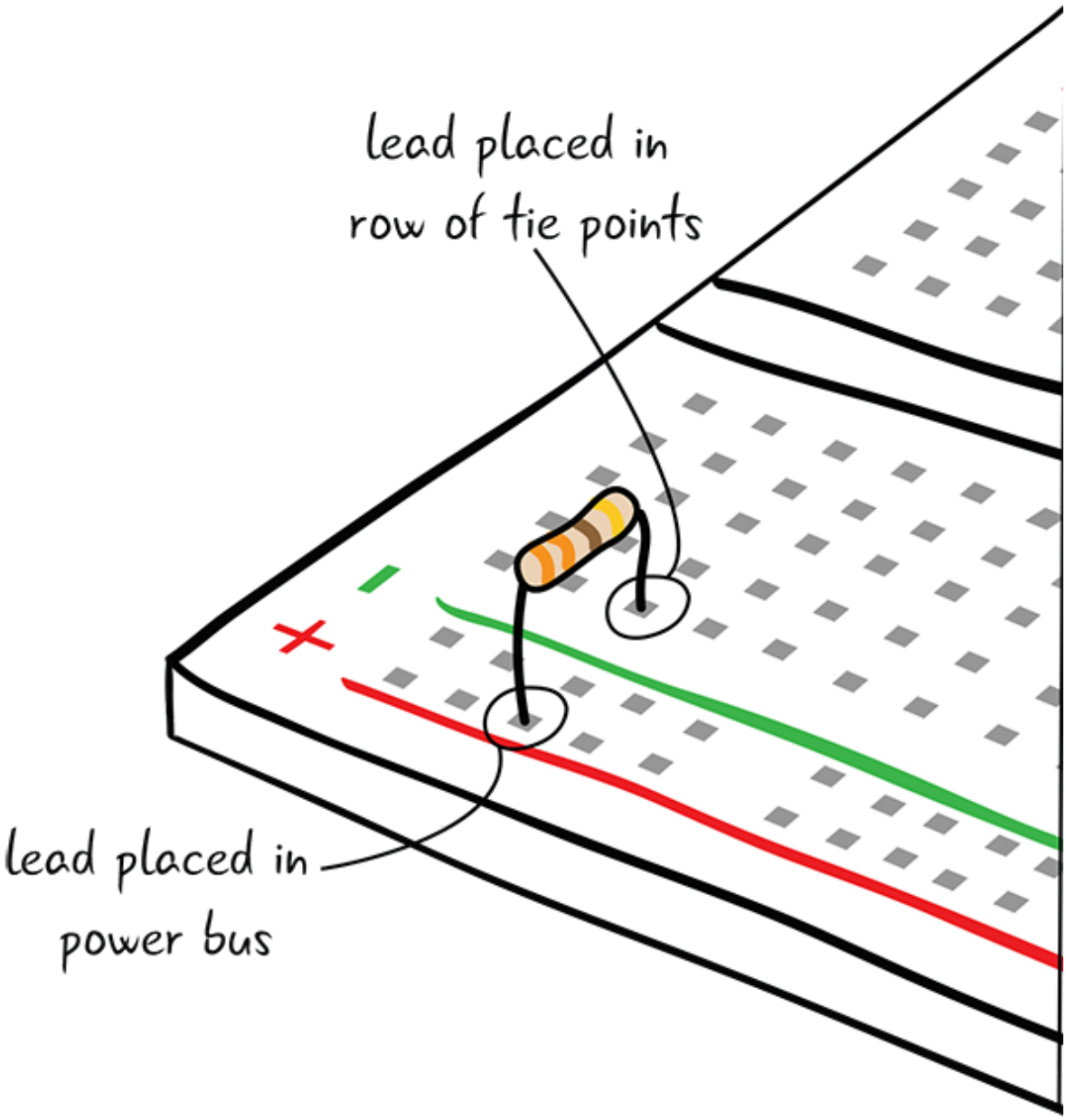


FIGURE 3-21: First add the resistor.

Next add an LED (Figure 3-22). The anode (long lead) goes in the same row of tie points as the resistor. The cathode (short lead) goes into the next row.

Figure 3-23 shows how one end of the resistor is in the same row of tie points as the anode of the LED.

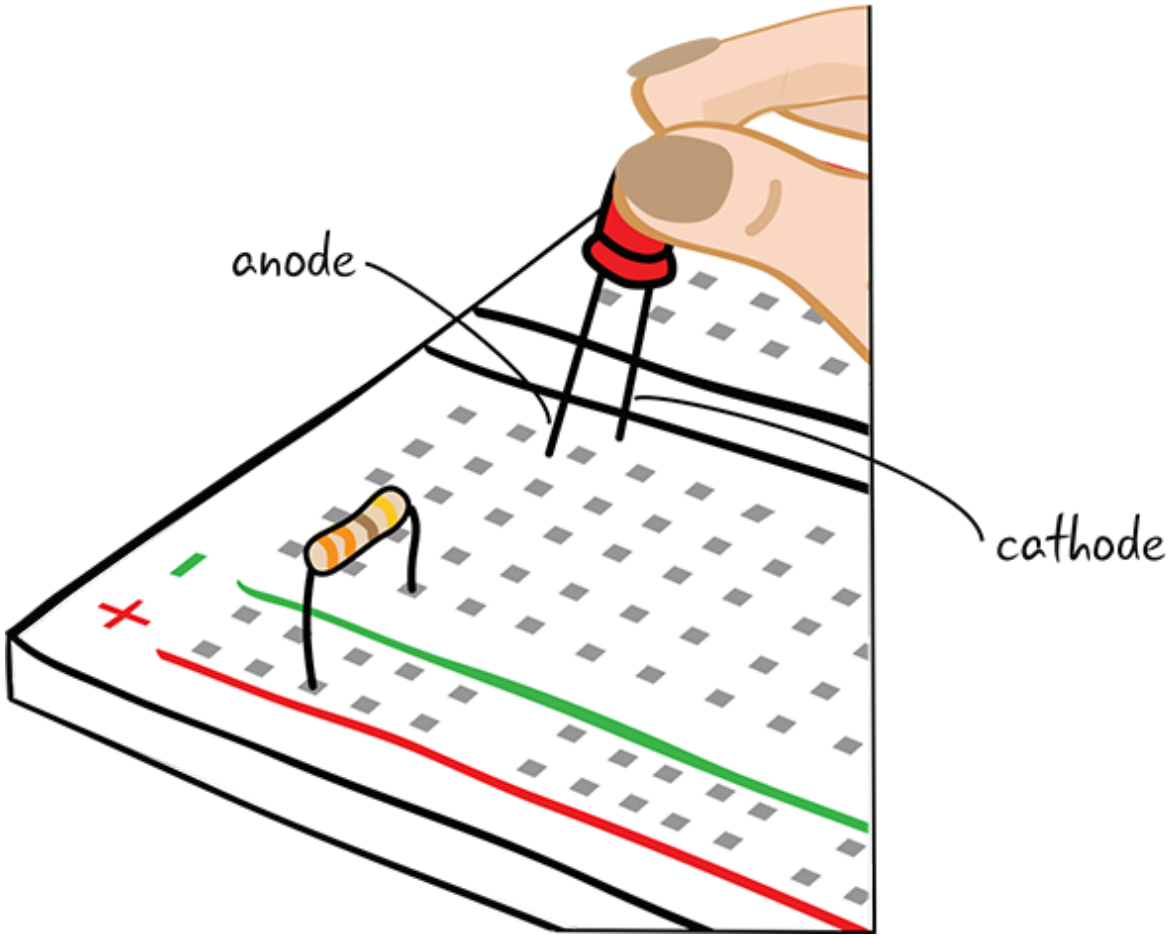


FIGURE 3-22: Add the LED.

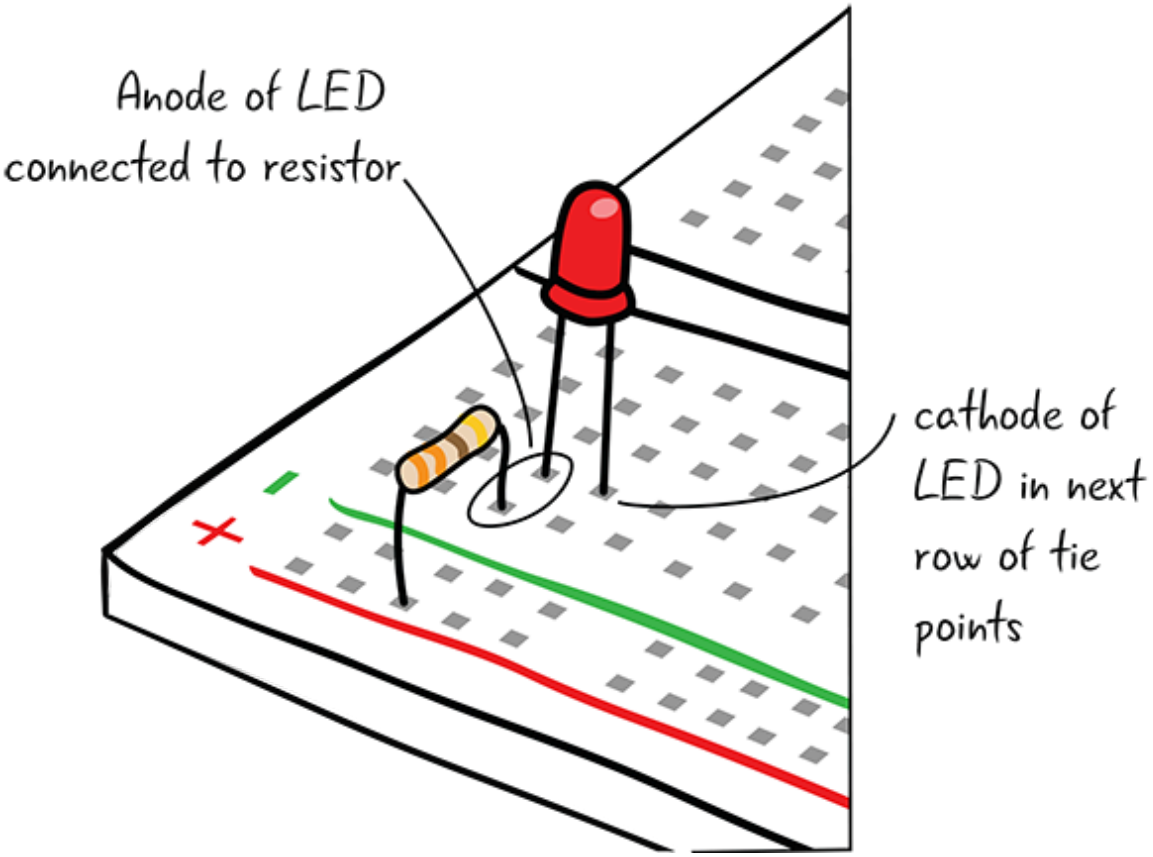


FIGURE 3-23: LED placed correctly

Next, you should put a jumper connecting the ground bus (marked with the – sign) to the cathode of the LED, as shown in Figure 3-24. Using a black jumper will indicate that it is going to ground. The jumper is just there to make a connection between the cathode and the ground bus.

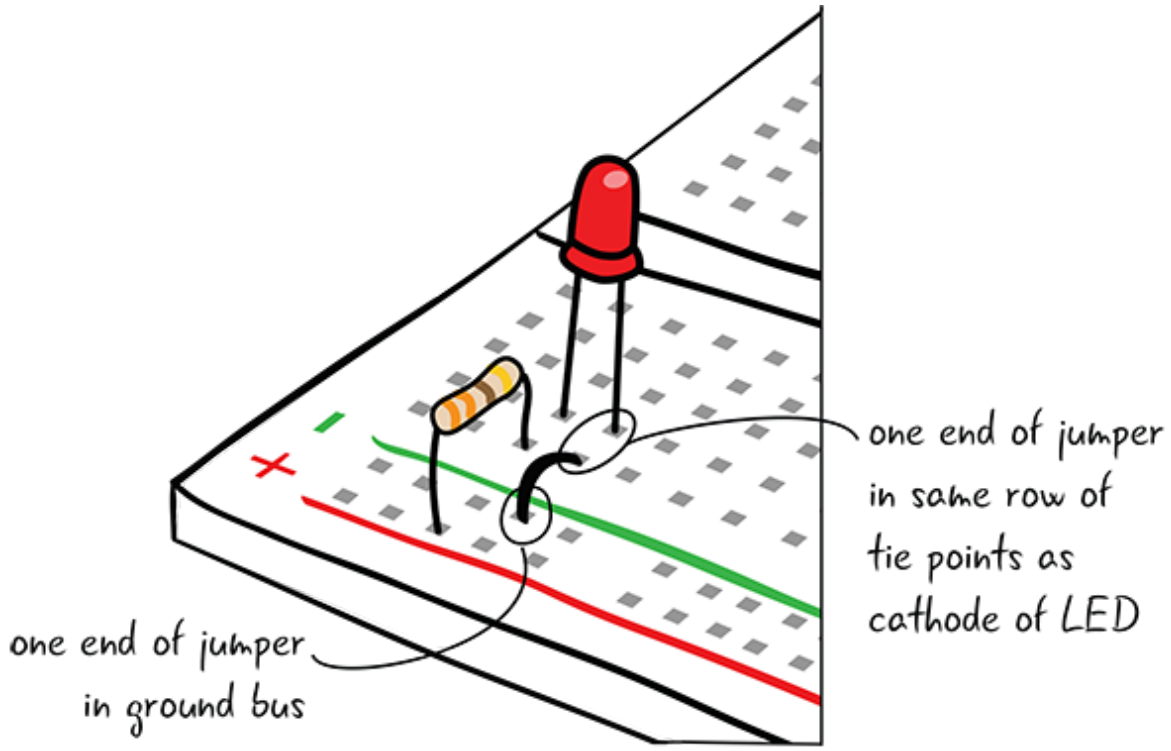


FIGURE 3-24: Add a jumper to ground.

Add the battery cap into the breadboard power and ground bus (Figure 3-25). It has metal ends that will fit into the power and the ground bus.

> **Tip**
>
> Make sure you get a secure fit into the breadboard. Doing so can be tricky; sometimes twisting the wire at the end of the battery cap can help.

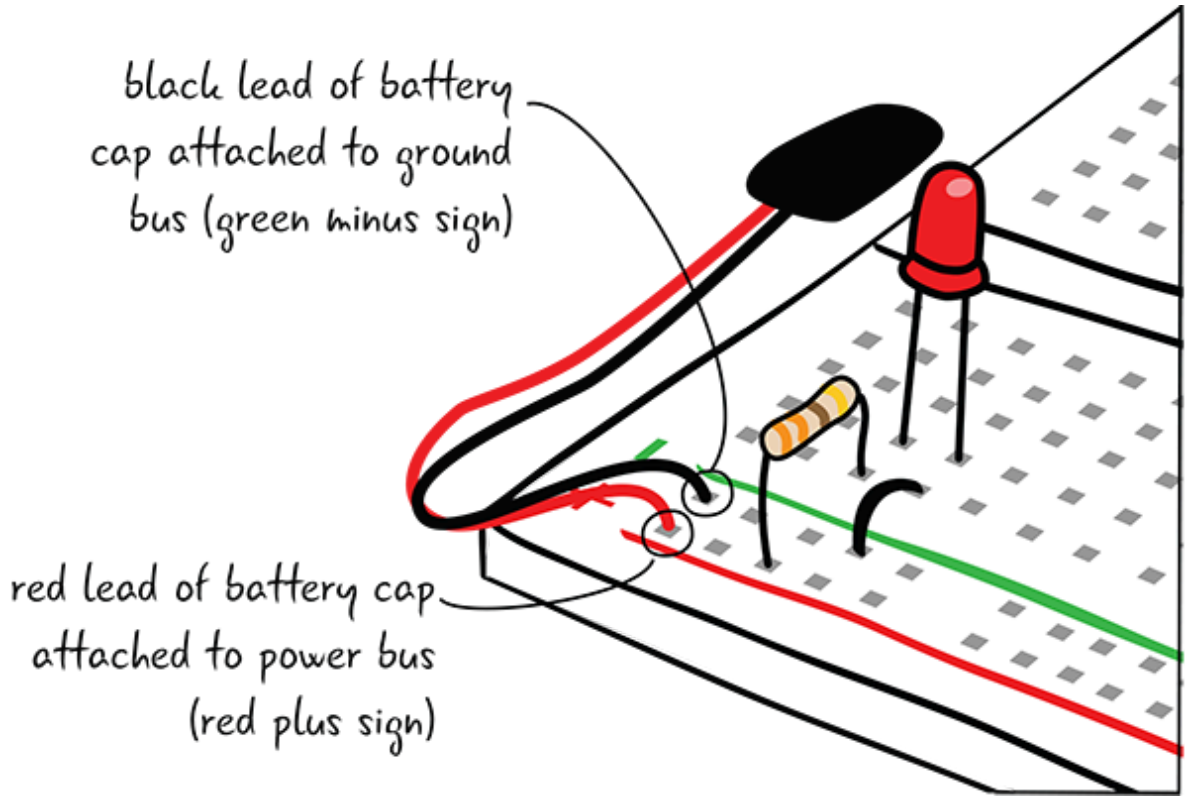


FIGURE 3-25: Add the battery cap to the breadboard.

A LOOK AT THE BATTERY

Let's take a closer look at the 9V battery and the battery cap. The top of the battery has two terminals that attach to the snap connectors on a battery cap, as shown in Figure 3-26. The smaller one, next to the plus (+) sign, is the power terminal. The larger terminal, next to the minus (–) sign, is the ground terminal.

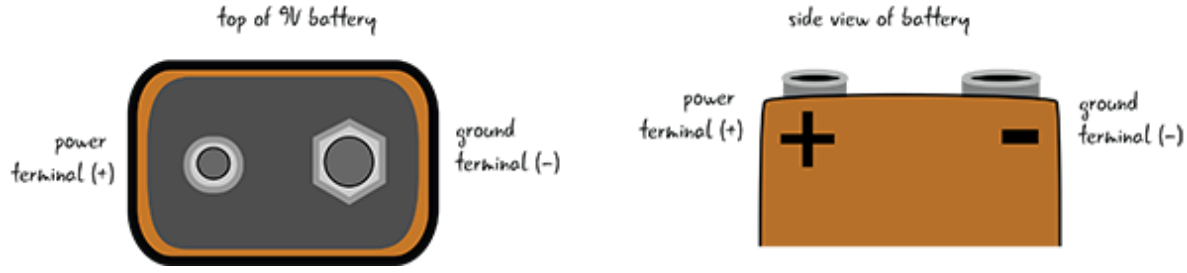


FIGURE 3-26: 9-volt battery up close

Turn over the battery cap, and look at the two snap connectors. The small connector will attach to the ground terminal and the large connector will attach to the power terminal, as seen in Figure 3-27.

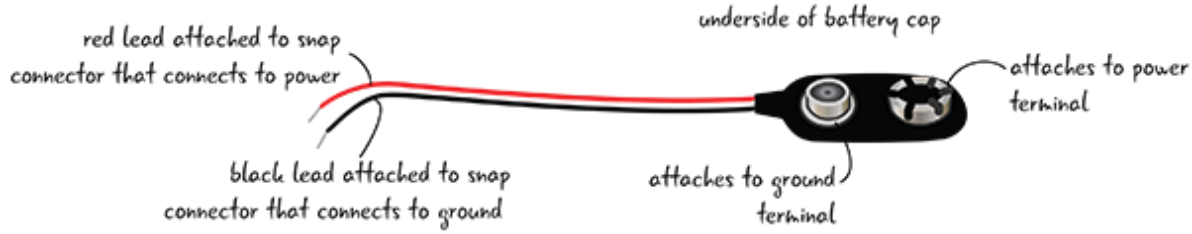


FIGURE 3-27: The battery cap

The snap connectors will only attach properly if the battery is correctly oriented, as shown in Figure 3-28. Your battery cap or holder may look different, but it will follow the same conventions.

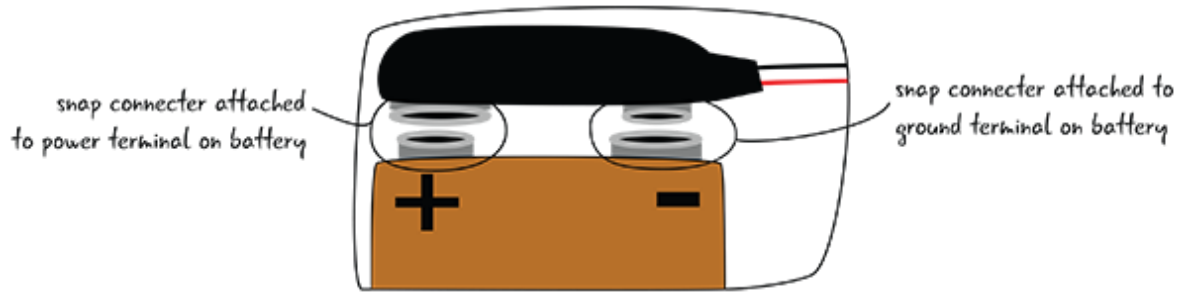


FIGURE 3-28: Attaching cap to battery

LET THERE BE LIGHT!

Now attach the battery to the cap. Your LED should light up (Figure 3-29). You've made your first circuit!

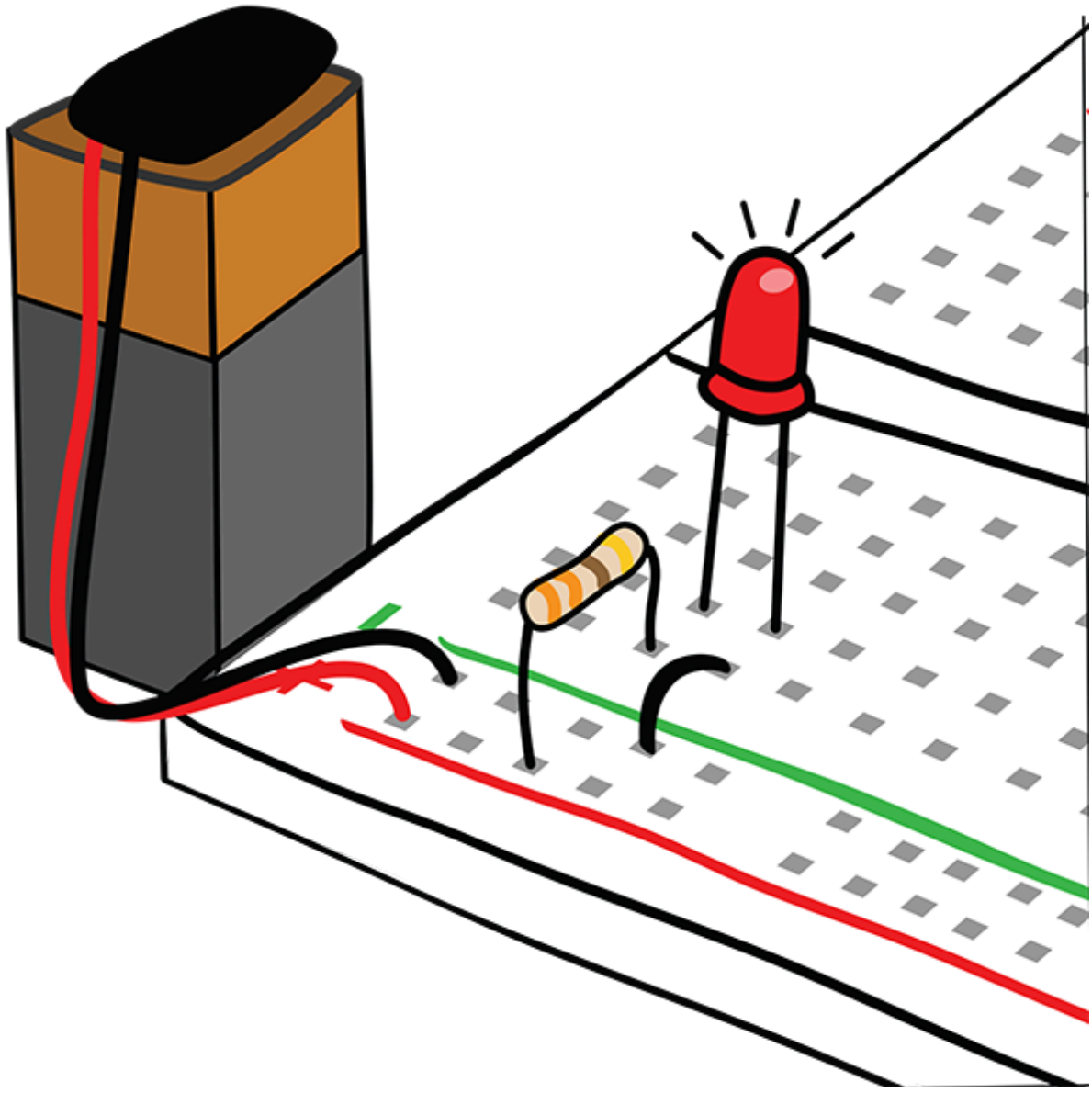


FIGURE 3-29: Your LED lights up!

This is just the first of many LEDs in the book, but feel good that you have turned on this first one. Next, let's look at how the battery is providing power to our circuit.

QUESTIONS?

Q: What if I don't have the resistor that you suggest?

A: We recommend buying a wide range of resistors initially to make sure that you have all of the resistors suggested for the first few projects and chapters in the book. Although there are ways to combine resistors to change their value, we don't cover them in detail in this book. It is generally best to have a variety to begin with.

POWER FOR OUR CIRCUIT: ELECTRICITY

The term *power* has a specific meaning when talking about electricity, which we'll explain later. For the moment, power here refers to the fact that electricity comes from our battery, passes through the resistor to the LED, and lights it up. Let's take a closer look at how this is indicated both on our battery and with the color of the wires in our circuit. We looked at the plus and minus symbols on the battery briefly when attaching the battery cap; now we'll look at the symbols in more detail.

A WORD ABOUT POWER SYMBOLS

As you can see in Figure 3-30, there is a + (positive) side and a – (negative) side on a battery, the conventional symbols used to mark which side of the battery produces power (positive) and which side is the ground (negative) side. (And you've seen the plus and minus signs on the buses on the breadboard.) You also saw that the positive side of the battery attached to the red lead on the battery cap and the negative side attached to the black lead on the battery cap.

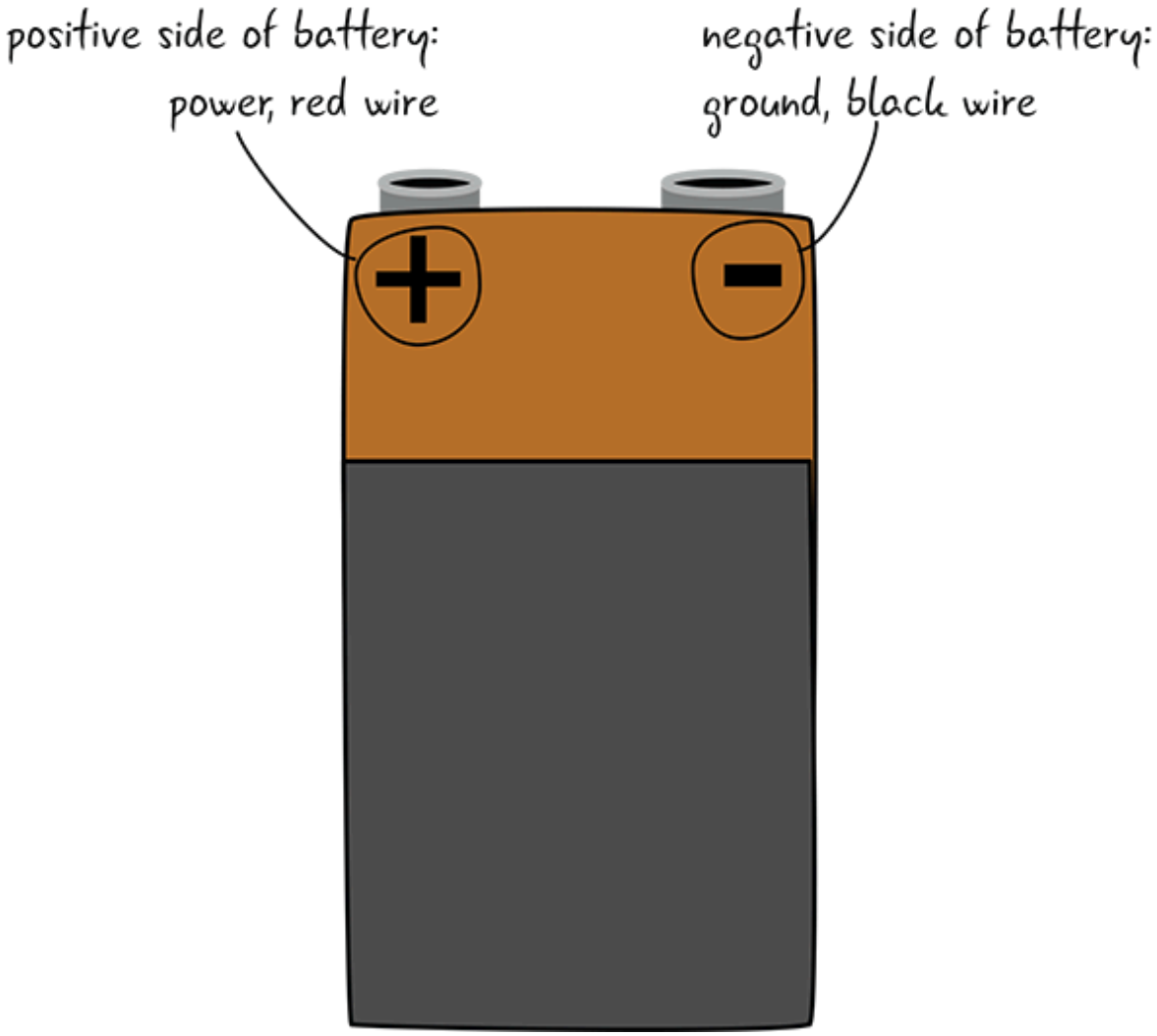


FIGURE 3-30: Positive and negative sides of a battery

Power

The + sign, or the positive, marks the power side of your battery. The convention is that power flows from this side of the battery, and all paths in your circuit must trace back to the power side. Standards also state that all wires that are connected to the positive side are red. This way, anyone who needs to look at or repair your circuit can immediately tell from where the power enters your circuit.

Ground

The – is the negative side of the battery, also known as the ground side. Just as all paths in the circuit must begin with the power side, they must all end at the ground side if you trace them along the entire length. Ground can be thought of as the "zero" side, the place where all power has been used up. All wires that lead back to the ground part of the circuit should be black; that will make it easier to work on your circuits and to know at a glance what parts are connected to ground.

We have looked a bit at power and ground, and you built your circuit. But what if your LED didn't light up? What steps can you take to find your problem and fix your circuit?

QUESTIONS?

Q: Do I need to use a new battery to light my LED? Can I use an old battery I found/borrowed around my house?

A: Yes, you can, but chances are your lights won't shine as bright as when you use a new battery. Batteries run out of power over time.

DEBUGGING THE CIRCUIT

Something went wrong or doesn't work right? What if the LED didn't light up? What might be wrong? Debugging!

Checking your circuit to see what is wrong is called *debugging*. Debugging is not just about solving the immediate problem, but also about creating a checklist of possible issues and solving them one by one. Sometimes the "obvious" solution is the hardest to find, and by following a checklist, you're sure not to miss anything.

ARE POWER AND GROUND CONNECTED TO THE BREADBOARD?

Make sure you connected the leads from the battery cap correctly to the power and ground buses on the breadboard, as shown in Figure 3-31. Remember: Connect the red lead to the bus with the red line next to it with a plus (+) sign at the top and the black lead to the ground bus with a green, blue, or black line (depending on your breadboard) and a minus (–) sign at the top of the breadboard.

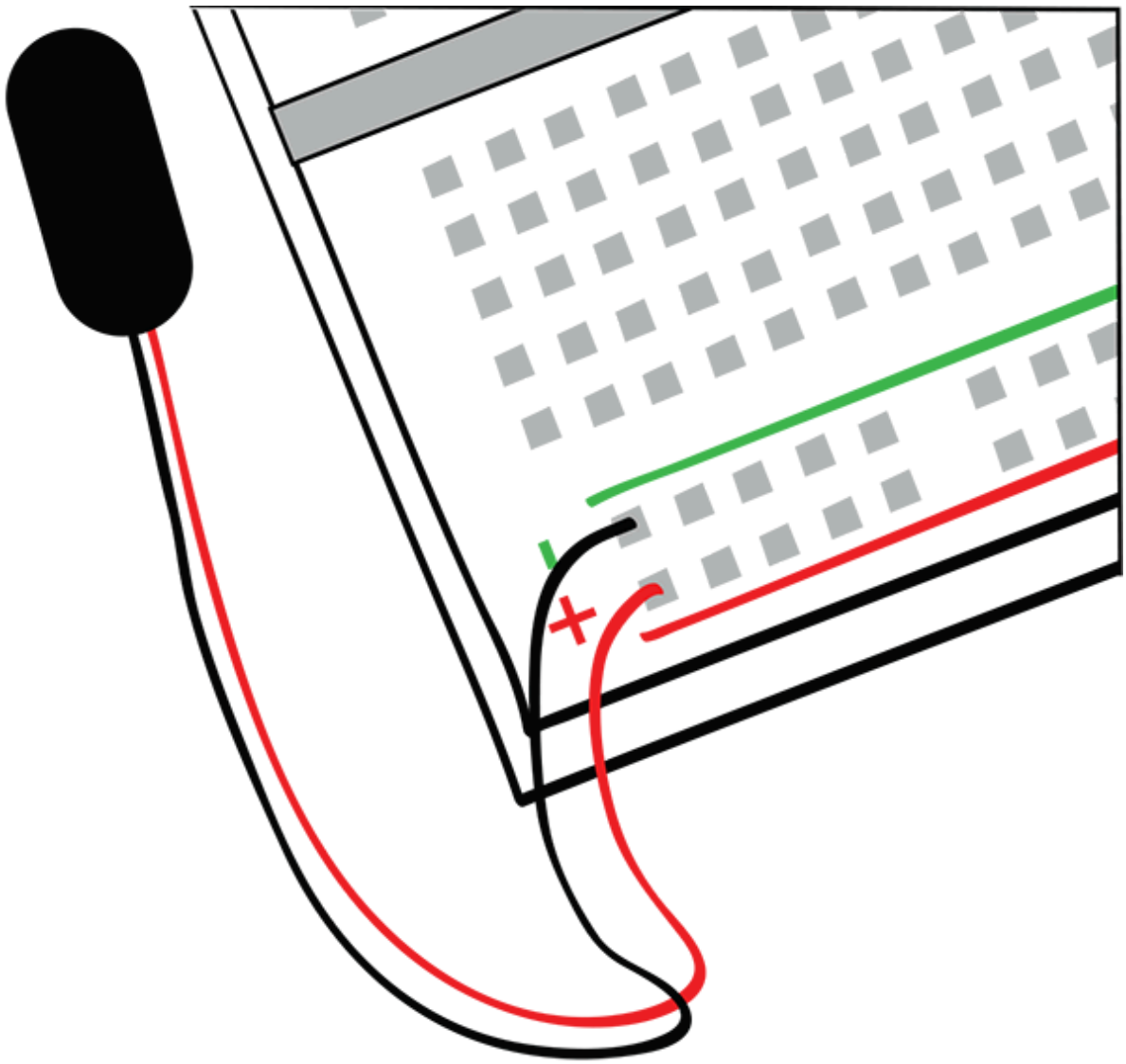


FIGURE 3-31: Leads from the battery cap attached properly to the power and ground buses

IS THE LED ORIENTED CORRECTLY?

Check to make sure you have placed the LED correctly on the breadboard. Remember it has a positive lead (anode) and a negative lead (cathode) and current flows through only if the LED is oriented correctly. The positive lead is longer than the negative lead, as shown in Figure 3-32.

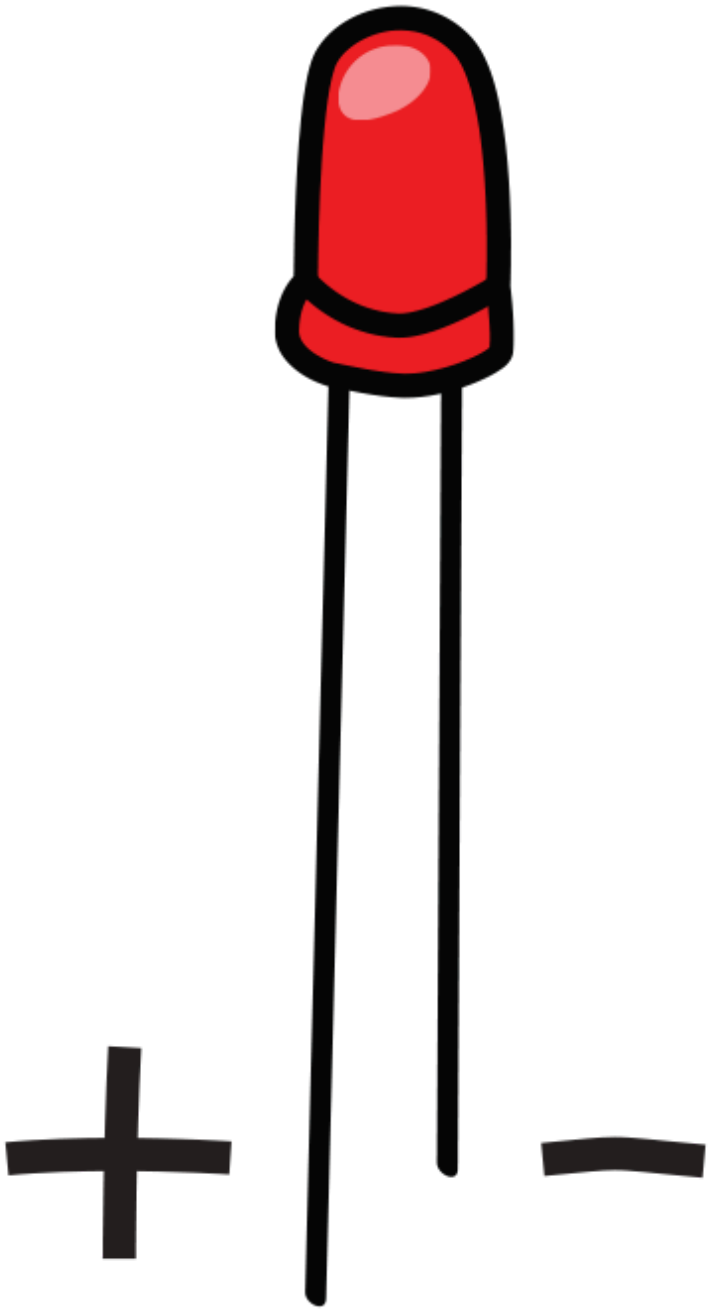


FIGURE 3-32: Positive (anode) and negative (cathode) leads of LED

DID I USE THE CORRECT RESISTOR?

Next check to see if you used the correct resistor. We'll discuss how to select a resistor in later chapters, but if you have used one with too much resistance, the circuit will not have enough power to light

up. If you use one that doesn't have enough resistance, you can destroy your LED. For this circuit, the resistor should have orange, orange, brown, and gold color bands (Figure 3-33).

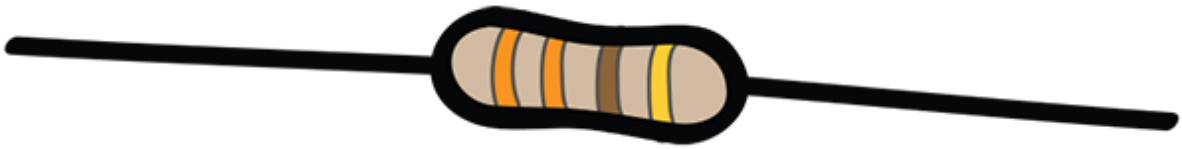


FIGURE 3-33: A 330-ohm resistor

These first few debugging steps rely on careful observation and understanding of the circuit basics we have covered so far. Some debugging steps will also rely on tools to enhance your knowledge about what happens in the circuit.

DEBUGGING CIRCUIT LOOPS: CONTINUITY

Perhaps the most common error in building a circuit using a breadboard is putting the components in the wrong tie points on the breadboard so they are not connected. As you've seen, circuits are loops, and if the components are not attached to each other properly, the loop is broken. *Continuity* is the property that simply means that things are connected, as shown in Figure 3-34.

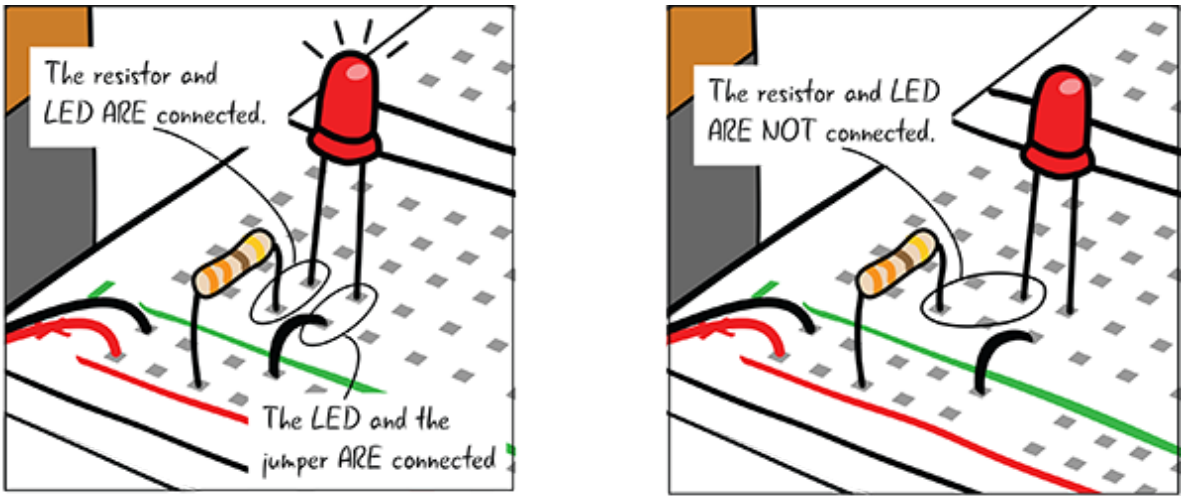


FIGURE 3-34: Components that are properly connected, and components that are *not* properly connected

You can check to see whether your components are attached correctly by looking closely at your board. Check carefully that the leads for the LED, resistor, and jumper are in the correct rows of tie points on the breadboard so they are connected properly.

There is another way to test for continuity in a circuit on a breadboard besides inspecting it visually: you can test for continuity with a *multimeter* (Figure 3-35).

FIGURE 3-35: A multimeter

QUESTIONS?

Q: Will I have to memorize the steps for debugging?

A: Good question. You are not required (or expected) to memorize the debugging steps. You'll find that after building the circuits in the book you'll begin to remember the debugging steps since you'll use them frequently. We'll reference the steps as need be for the remainder of the book.

THE MULTIMETER

Another way to find out information about your circuits is by using a multimeter. A multimeter is a critical tool for verifying that our electronic and Arduino projects are running correctly and that all of our parts are functional. Your multimeter will be a great tool to use throughout the projects in this book to ensure everything is working as expected. We'll sometimes call it a multimeter, and sometimes we'll refer to it as a meter. Now we'll show you how to use it to test for continuity.

You won't use a multimeter with the Arduino here, but you will in future chapters. Why are we looking at it now? It will help you debug your first circuit, and it will be invaluable later on when your projects become more complex and you learn more ways to use it. Figure 3-36 shows a few different multimeters.

We are using the meter from SparkFun (SparkFun part number TOL-12966), which we mentioned in the parts list in Chapter 1. The drawings of the multimeter in this book are all based on this model. Your meter may look different, but the principles of setting up the meter and using it will be the same.

FIGURE 3-36: Multimeters come in different sizes and colors.

MULTIMETER OVERVIEW

Figure 3-37 shows the parts of a multimeter: a *display* that shows the value of the electrical property you are measuring, and a *dial* that turns to determine the electrical property you are testing. One end of the *probes* touches the components you are testing at one end, whereas the other end is attached to the meter in the *ports*.

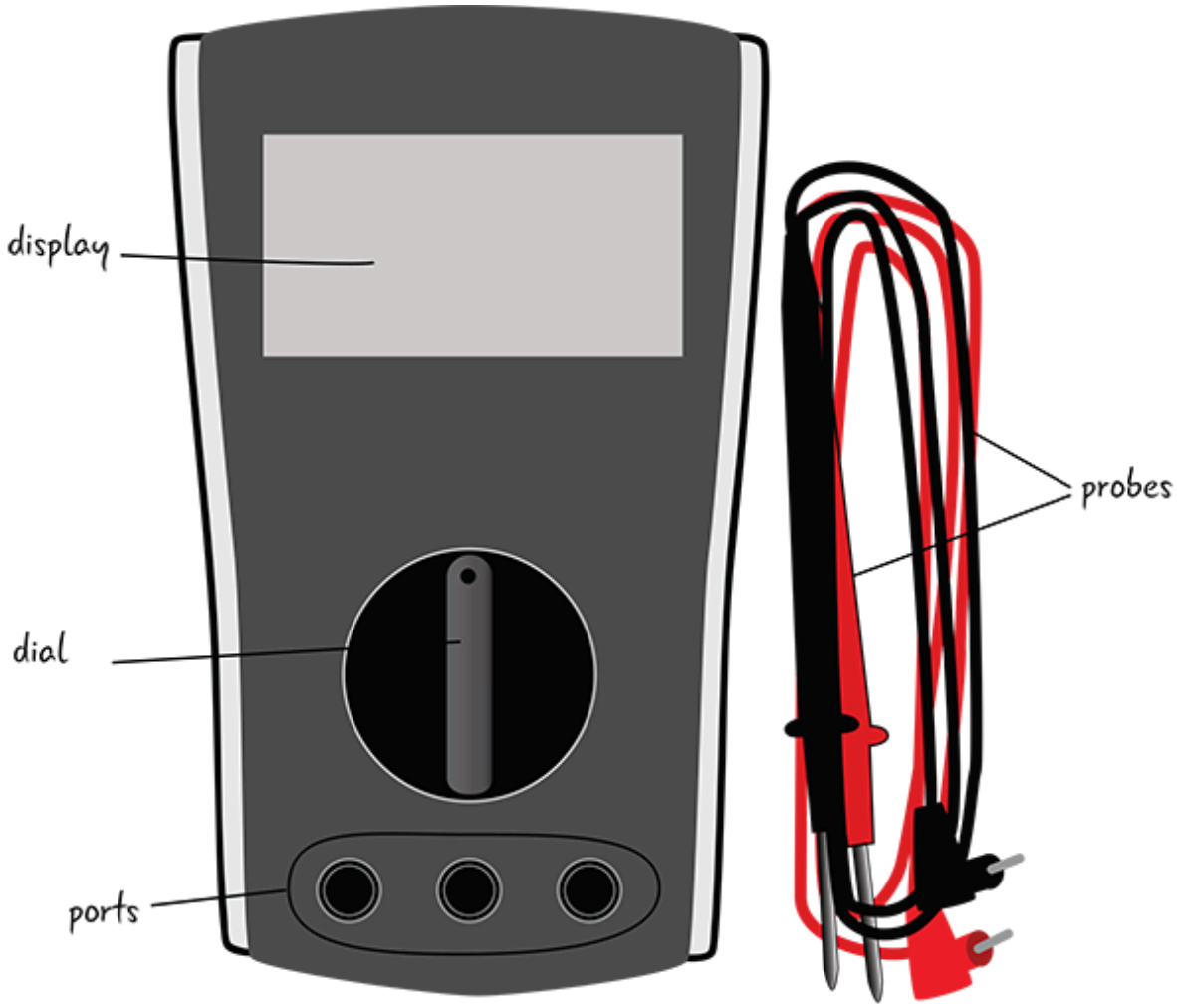


FIGURE 3-37: Parts of a multimeter

Some meters have off /on buttons, whereas this one turns on with the dial.

Warning

Remember to turn your meter off when you are done so you don't run down the battery.

Most multimeters are powered by a 9V battery. We don't include instructions for inserting the battery into your meter. If you get this

meter, the instructions will come with it. If you purchased or inherited a different meter, the instructions for replacing the battery will be different.

PARTS OF THE MULTIMETER: THE DIAL

Figure 3-38 is a detail of the dial of a typical multimeter marked with some of the electrical quantities it can measure. We'll explain all these symbols and properties as we progress through the book. Right now just know that there are different properties you can measure: AC voltage, DC voltage, resistance, DC amperage, and continuity.

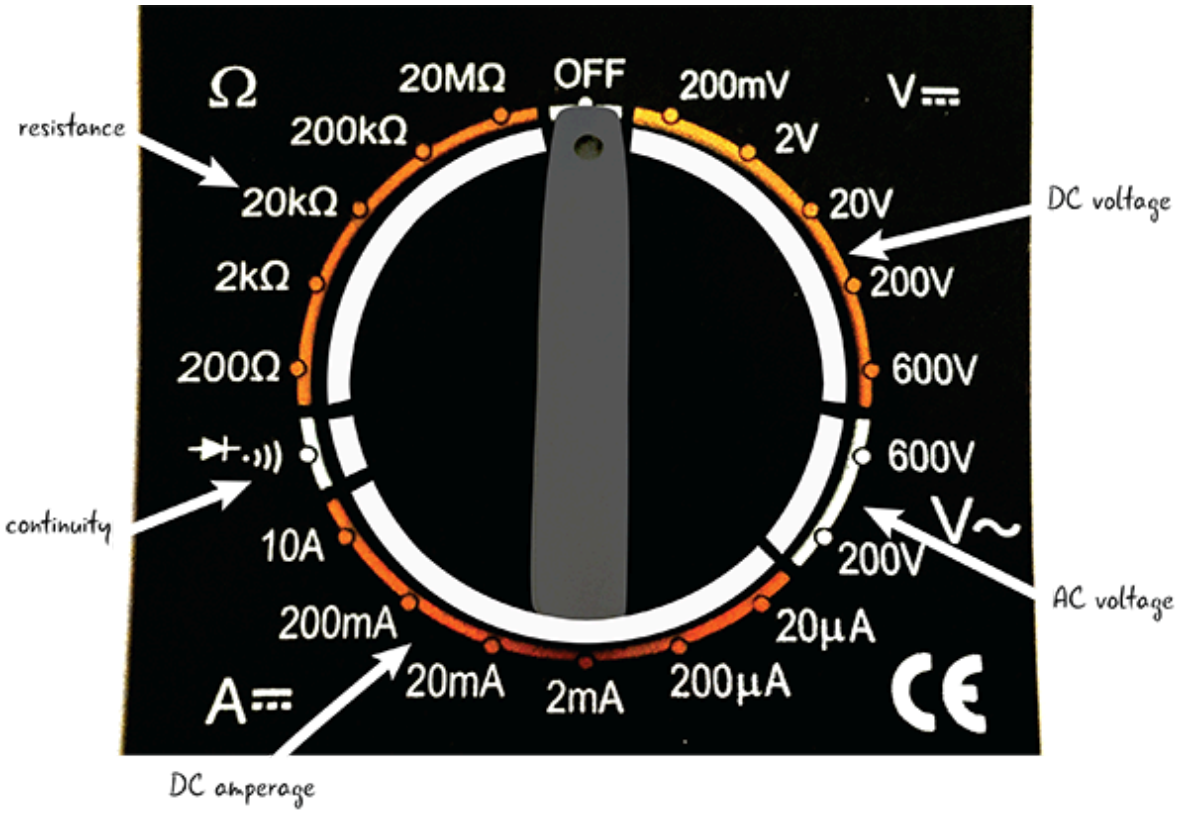


FIGURE 3-38: The dial of a multimeter with electrical properties it can measure

We'll return to these electrical properties and how to measure them with the meter in Chapter 5, "Electricity and Metering."

PARTS OF THE MULTIMETER: THE PROBES

Figure 3-39 shows the probes, the part of the multimeter that touches your circuit, component, or whatever it is you're testing or measuring. The metal tips of the probes are placed so that they touch the circuit or component. The other end of each probe snaps into the ports on the multimeter. The probes will not be attached to the ports when you unpack the multimeter.

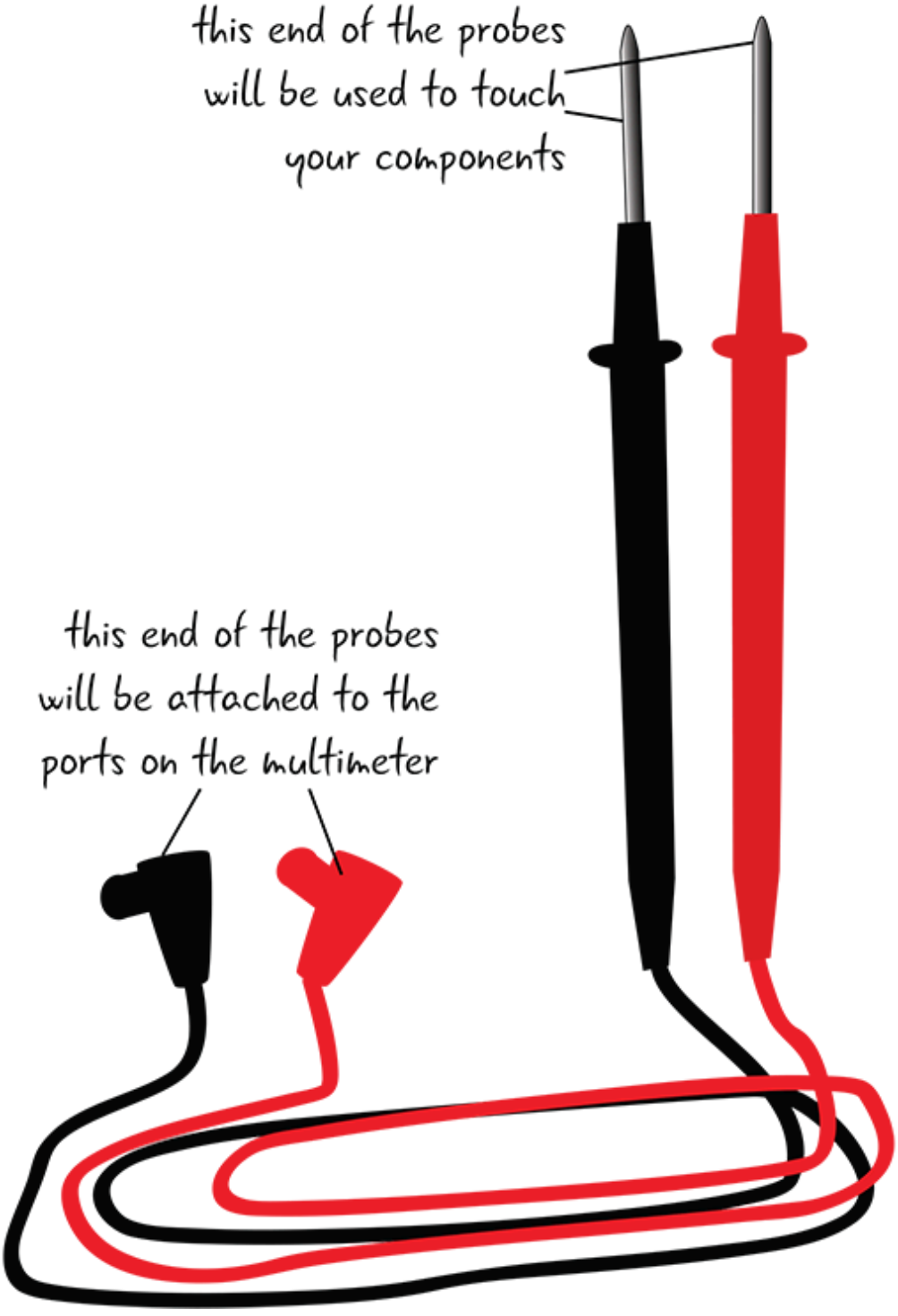


FIGURE 3-39: Probes of the multimeter

PARTS OF THE MULTIMETER: THE PORTS

Now that we've looked at the probes of the multimeter, let's take a closer look at the ports on the meter, shown in Figure 3-40.

It is important that the probes be placed in the correct ports when using a meter. For all measurements, the black probe is placed into the center COM port. The red probe has two different ports in which it can be placed (the outsides as marked). Generally, keeping the red probe in the far-right port is a good practice.

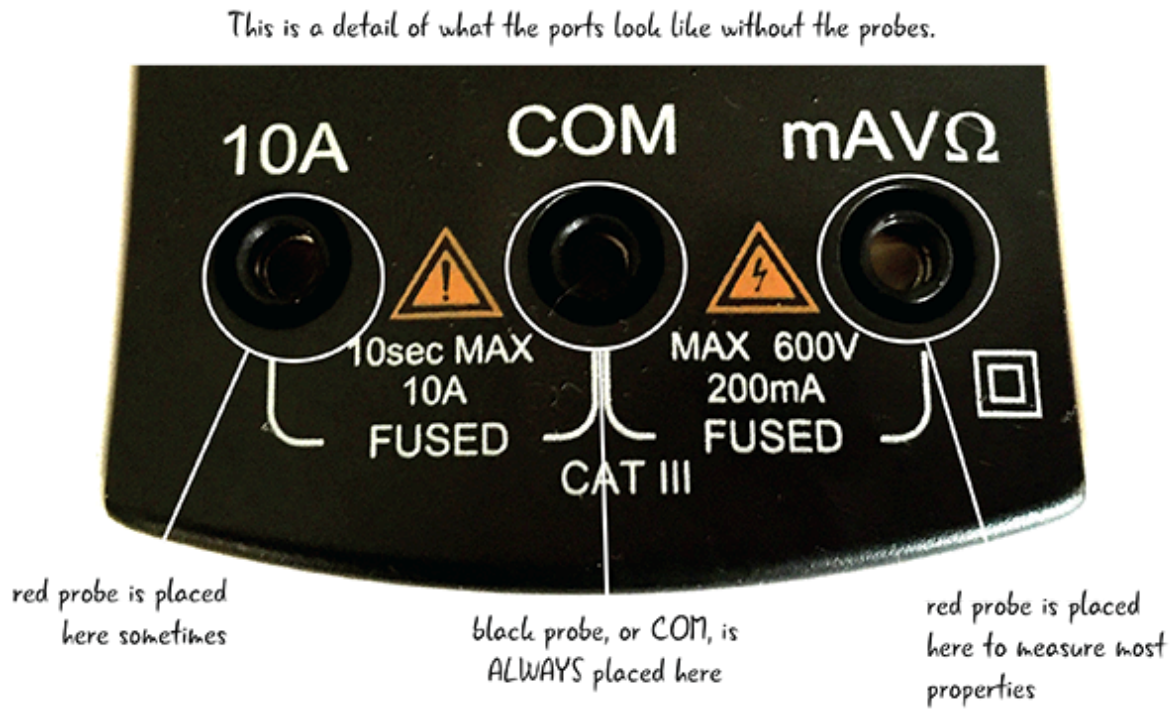


FIGURE 3-40: The ports on a multimeter

USING THE MULTIMETER

Continuity (Figure 3-41) is an electrical property that shows whether a connection exists between parts. You can use the meter to test this property. It's a good way to get familiar with the parts of your meter. And you're going to use it for debugging your circuit!

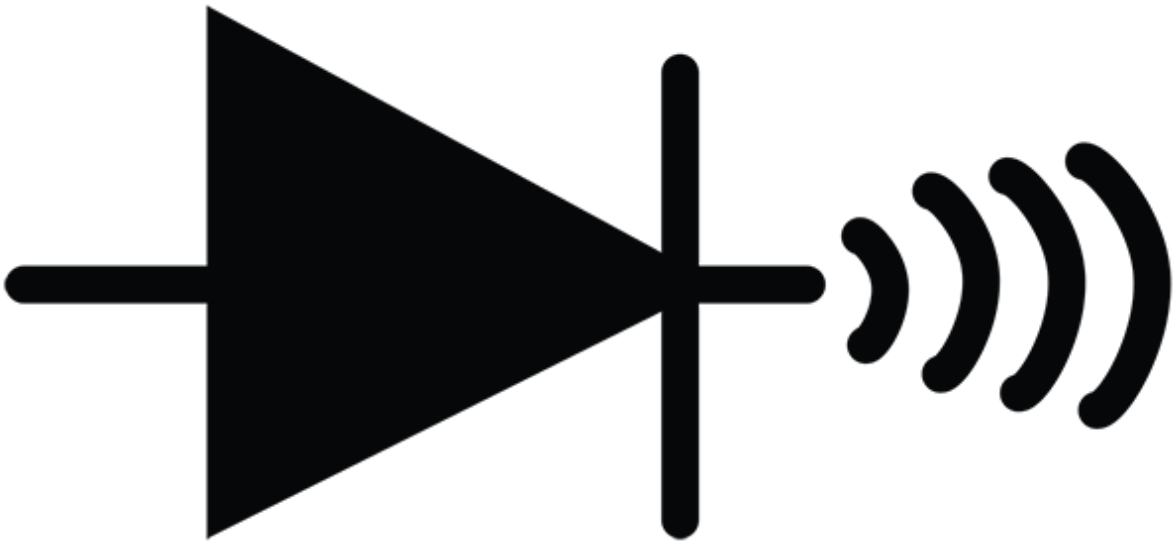


FIGURE 3-41: Symbol for continuity

SETTING UP YOUR METER TO TEST FOR CONTINUITY

First, we'll show you how to use the multimeter to test the electrical connection between the probes on the meter, checking the "continuity" between the probes (Figure 3-42). We'll then move on to testing continuity in your circuit.

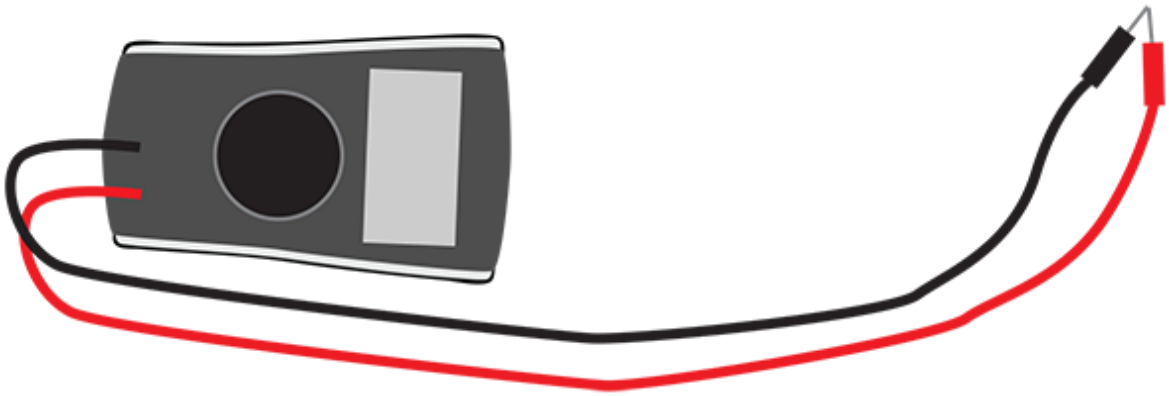


FIGURE 3-42: Multimeter with the probes touching

This test is a good way to make sure that your multimeter is functioning and to get familiar with how to use it. If the probes are touching, they form a complete electrical loop. The same test can be

used later to check if your parts are connected correctly from an electrical perspective.

METER SETTINGS FOR TESTING CONTINUITY

To test continuity, the black probe goes in the port marked COM and the red probe goes in the port marked mAVΩ, as seen in Figure 3-43.

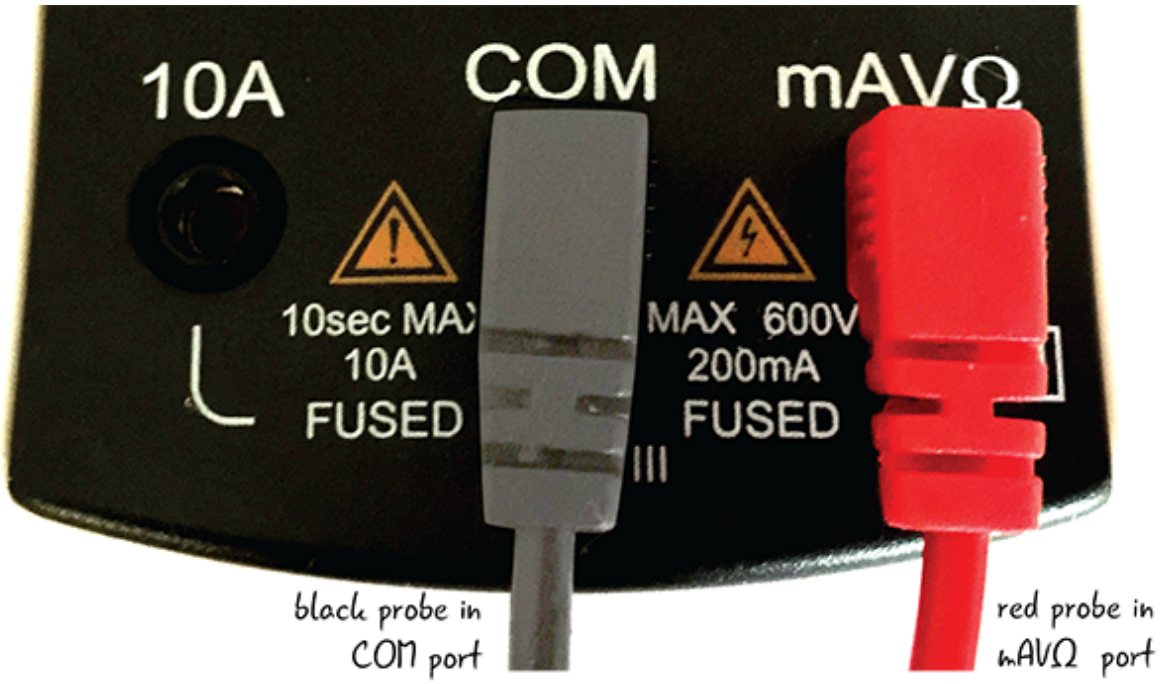


FIGURE 3-43: Meter port settings to test for continuity

Next, move the dial so the knob is pointing toward the continuity symbol (Figure 3-44).

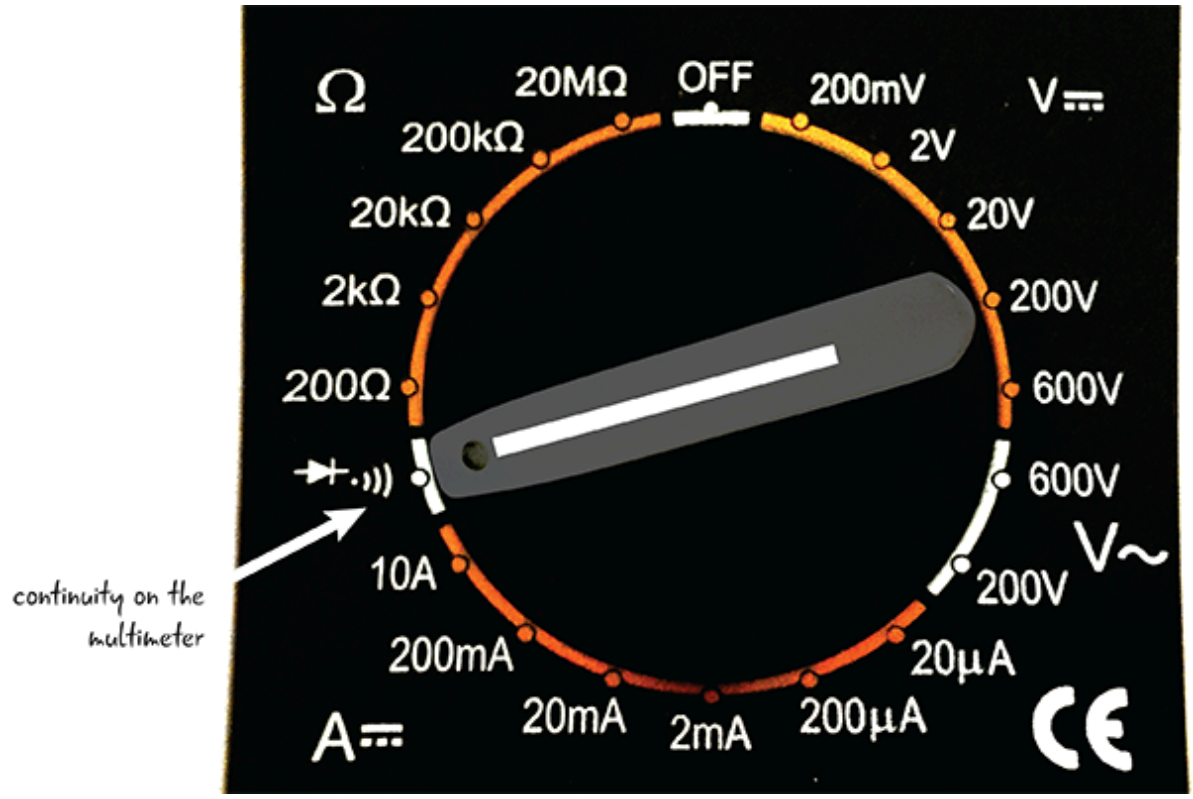


FIGURE 3-44: Turn the knob to the symbol for continuity.

TESTING CONTINUITY

When the probes touch components that are connected, the meter will play a tone if the meter is set to test for continuity. When the probes are attached to the ports correctly, if they touch each other they make an electrical loop. You are making a circuit with your probes in order to test continuity.

Touch the two probes together now to test this out, as shown in Figure 3-45. While the probes are touching, the screen will display ".000," though it may fluctuate slightly. You'll also hear a tone that will vary in sound depending on your meter. For continuity, the display numbers are not as important as they will be with the other properties we'll talk more about in Chapter 5.

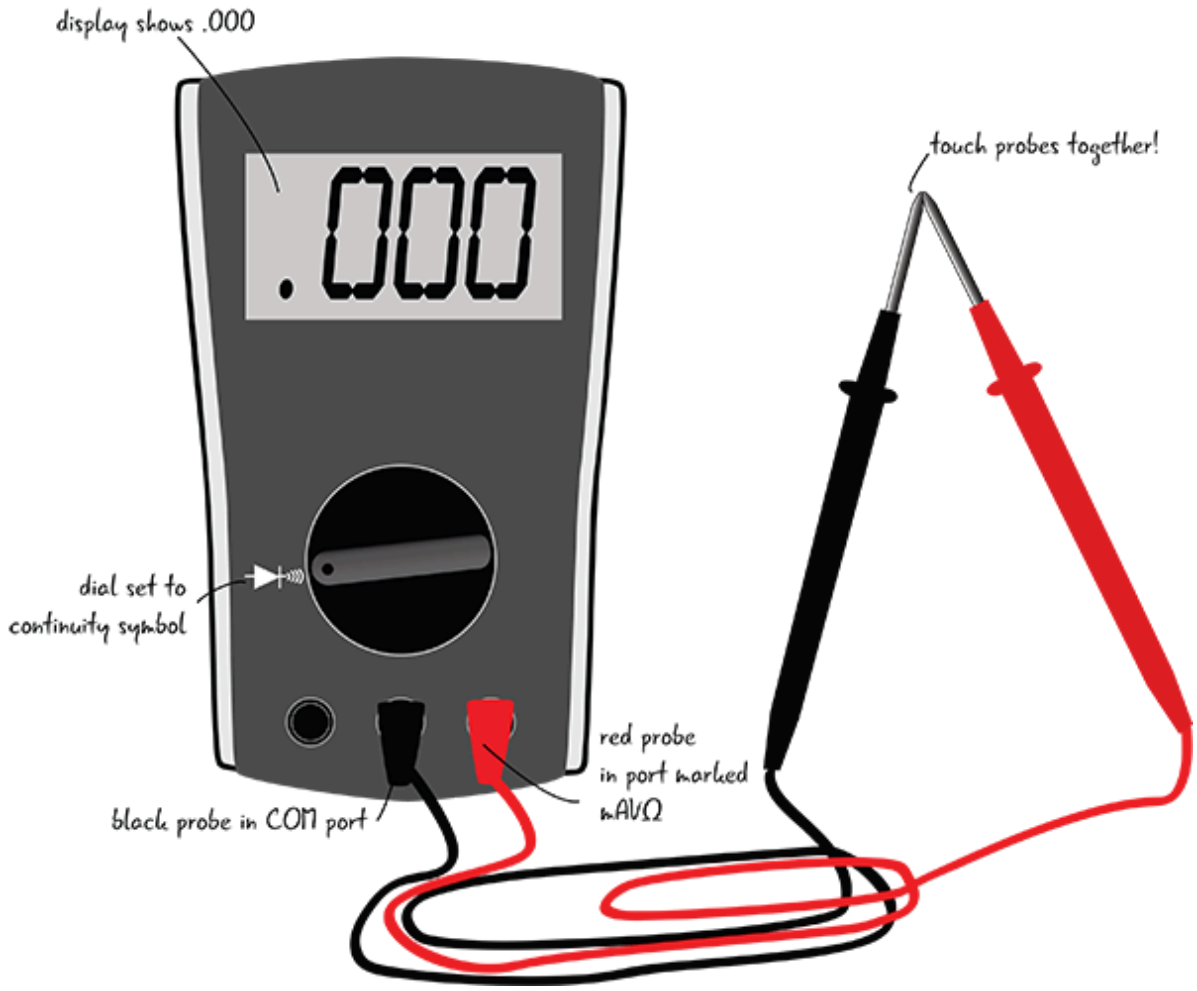


FIGURE 3-45: The multimeter with the probes touching is a test for continuity.

When the probes touch, as shown in Figure 3-46, you should hear a tone!

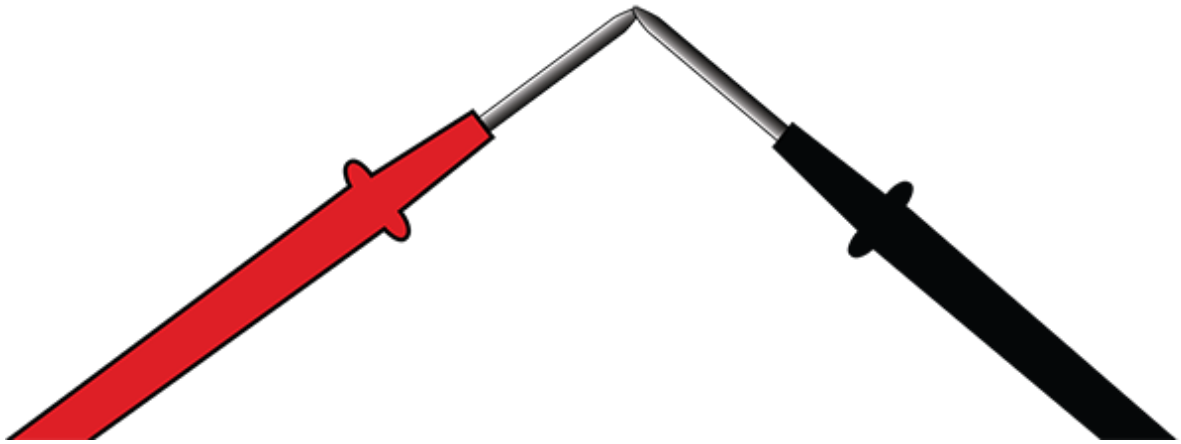


FIGURE 3-46: The probes touch, and a tone plays.

Continuity will help troubleshoot issues in more complicated circuits by identifying when components are not connected to each other or if they are connected in an incorrect spot. We'll show you more explicitly how continuity can help you solve issues in Chapter 5.

BACK TO DEBUGGING OUR CIRCUIT

Let's return to our basic circuit. Now that you have unboxed your multimeter and understand what continuity is, let's apply the multimeter probes to our circuit and take a look at our results.

TESTING CONTINUITY IN A CIRCUIT

Your meter is already set up correctly to test continuity if you just completed the last exercise. The settings for the dial and probes are shown in Figure 3-47. Check to make certain that the dial is set to the continuity symbol and the probes are in the right ports.

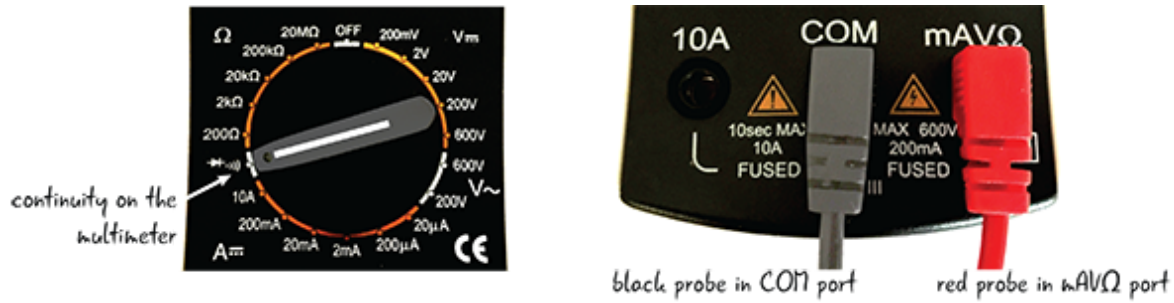


FIGURE 3-47: Meter settings to test continuity

First, remove your battery from the circuit. Then, turn on your meter and place the probes on one of the leads of the resistor and one of the leads of the LED, as shown in Figure 3-48. It doesn't matter which color probe touches which lead.

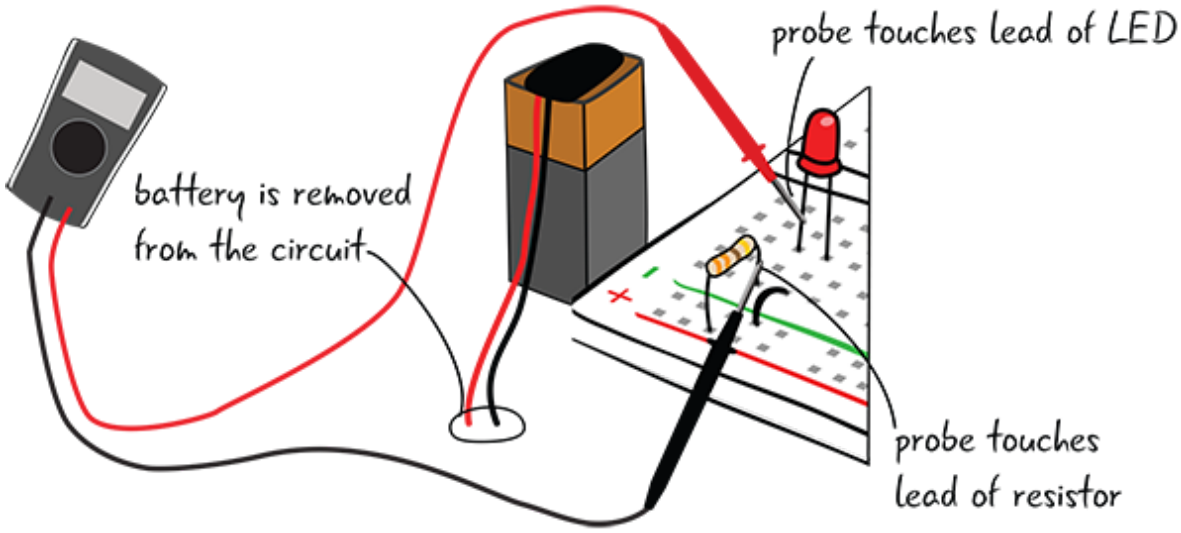


FIGURE 3-48: Testing the circuit for continuity

If your components are connected, you'll hear the buzzing sound again, and the settings on the display will read .000 with a possible slight fluctuation.

What if you didn't get that buzzing sound? Check the connections in your breadboard between each component to see if they are in the proper tie points.

In Figure 3-49, the LED is not connected to any of the other components. The resistor is attached to the power rail and the jumper is attached to ground, but neither is attached to the LED. To fix the circuit, put the leads in the correct tie points.

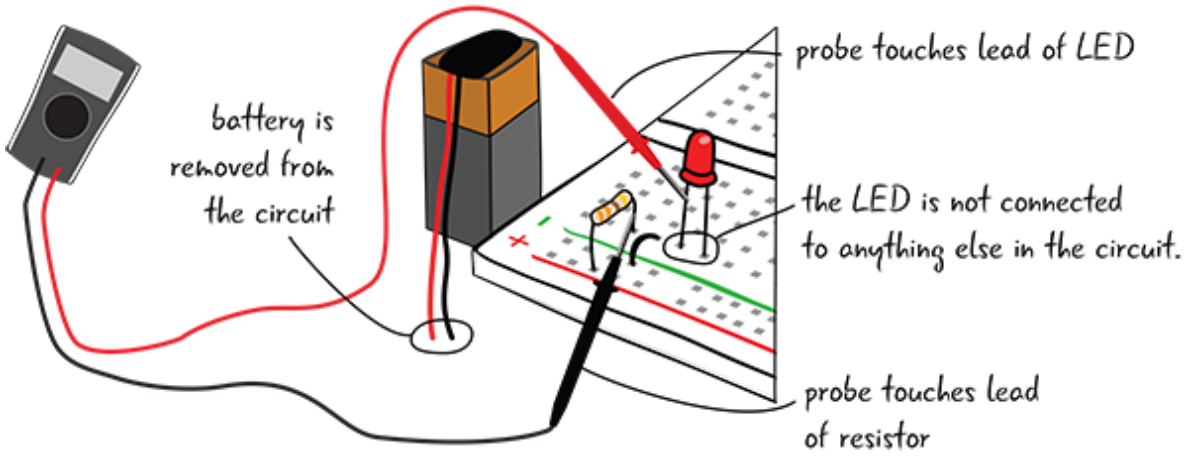


FIGURE 3-49: The multimeter testing a circuit where the components are not connected

QUESTIONS?

Q: What about all those other symbols on the multimeter—when will we use the multimeter to measure those?

A: We'll explain more about the multimeter and how to measure the various electrical properties (resistance, voltage, current) in Chapter 5.

Q: What if my meter has a different reading than .000 when I'm testing continuity?

A: With the recommended meter, the most important thing to pay attention to when checking continuity is to listen for the noise created by the meter that indicates your components are electrically connected. Meters without sounds will have other ways of indicating continuity on the display screen.

SUMMARY

In this chapter, you learned how to build a circuit and how to debug it. You were introduced to the multimeter, and you learned how to use it to check if all your components were connected. In the next chapter, you'll set up your Arduino to program it, and then connect it to a breadboard to begin using your Arduino to control components.

4

PROGRAMMING THE ARDUINO

In this chapter, you'll start to see how the Arduino controls electronics with the programs that you write. First you'll set up the software to program the Arduino on a computer; then you'll connect your Arduino to a breadboard. We'll show you how to build an SOS signal light using an LED. You'll learn basic rules about writing code and get familiar with writing code in the Arduino environment. For this chapter, you need to know how to hook up your Arduino to a computer and how to build a basic circuit on a breadboard.

ARDUINO, CIRCUITS, AND CODE: BRINGING EVERYTHING TOGETHER

This is your first opportunity to combine building circuits with basic programming. When you add programming and the Arduino to your circuit, you have more control over the circuit; your LED can flash on and off in different patterns. You'll learn how to program the Arduino and connect it to a breadboard to build a complex circuit in which the

timing of the components in the circuit is controlled by the series of instructions loaded onto the Arduino. To illustrate this, we'll show you how to create an SOS signal light with an LED that flashes on and off according to timing controlled by the Arduino.

From this point on, most of the projects will include the three parts shown in Figure 4-1: the code, the Arduino, and a circuit on a breadboard. We'll discuss the combination of all three elements and how they interact with each other in this chapter.

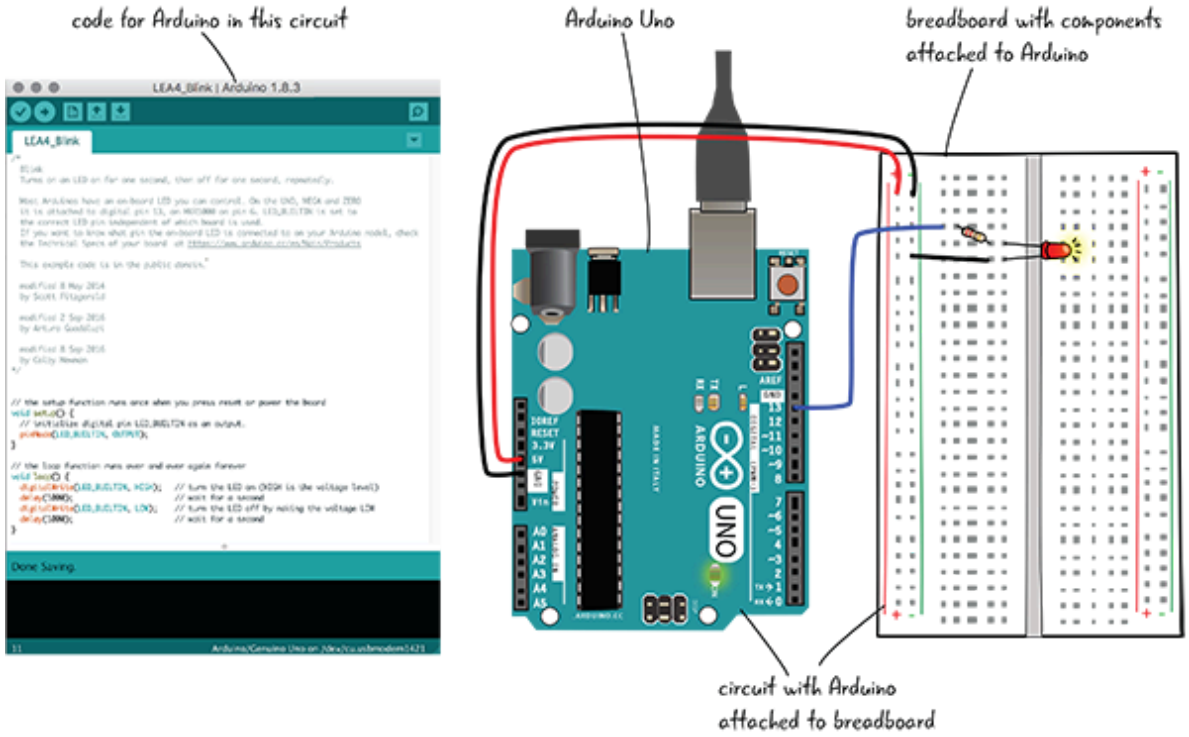


FIGURE 4-1: Code, Arduino, and the breadboard

We looked at the Arduino and some of its features in Chapter 2, "Your Arduino." In Chapter 3, "Meet the Circuit," you learned a bit about small-scale electronics and circuits. We'll walk you through downloading and using the Arduino IDE in this chapter, which will allow you to upload code, changing the behavior of the Arduino.

Just as we'll show you the necessary circuits throughout the book, we'll also include all the code examples you will need to run your projects.

To code, you'll need software from Arduino installed on your computer. You'll download and install the Arduino IDE. What's an IDE? Let's take a look.

WHAT'S AN IDE?

An integrated development environment (IDE) is a software application that allows you to write code and test that code out in the programming language the IDE supports.

If you have experience programming, you may have used another IDE to write, test, debug, and turn your code into something the computer understands. If you haven't, the Arduino IDE is a good place to start—it is relatively simple and easy to understand.

The Arduino team has designed an IDE for use with their devices that has all the features you need. It has a built-in code editor, which is a program used to write the text files that you create when programming. You can test your code in the IDE and solve any emerging problems with the help of a message area that shows errors in your code and a console that provides more detail about the nature of these errors. It has buttons so you can check your code, save it, create a new code window, upload it to your Arduino, and more. This matches nicely with the basic flowchart for Arduino projects as shown in Figure 4-2.

Note

Uploading is transferring the instructions you write in the code editor to the "brains" of the Arduino so that your code controls the Arduino.

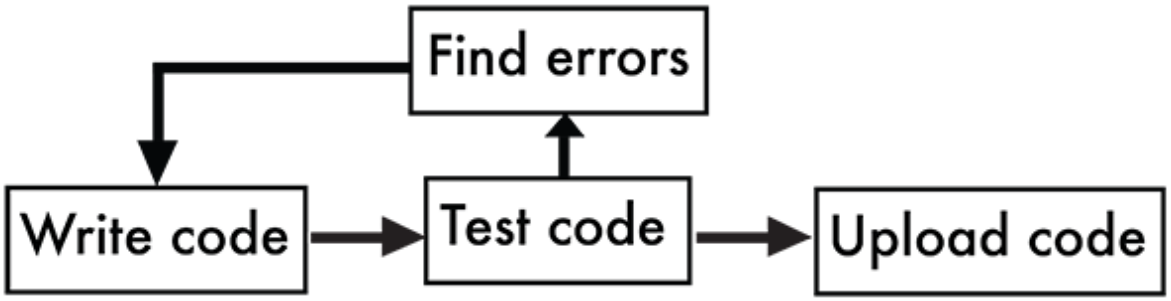


FIGURE 4-2: Arduino flowchart

The IDE is freely available on the Arduino website at arduino.cc/en/Main/Software. It is possible to program an Arduino using another text editor or IDE, but we'll stick with using the Arduino IDE in this book.

WHAT'S IN THE ARDUINO IDE?

So what's in the IDE?

A code editor window where you write your code

A message area that gives information about your code

An error console that gives detailed information and helps in debugging

Menus that allow you to set properties for your Uno and load code examples and other functions

Buttons to check code, upload it to Arduino, save code, create a new code window, and more

WHAT IS CODE?

In basic terms, *code* is used to give instructions to the computer. We use code to speak in the language the computer understands (in this case, the Arduino language) in order to accomplish a set of tasks or to set up a series of programmed responses. Computers have a hard time understanding what you mean, imply, or suggest. They are not

capable of the finer points of language, so we use code to simplify the instructions to a set of commands at a fundamental level.

You've seen what's in an IDE, as well as a basic description of code. Let's take a quick look at the Arduino IDE.

ARDUINO IDE: FIRST GLANCE

Here's your first look at the Arduino IDE. Don't worry about memorizing any of the parts or what they do—this is just a first glance. We'll cover all of the parts in detail later in this chapter and in the rest of the book.

As you can see in Figure 4-3, menus appear at the top of the window. There are also buttons for frequently used functions such as save, an area where you can write code, and some message areas.

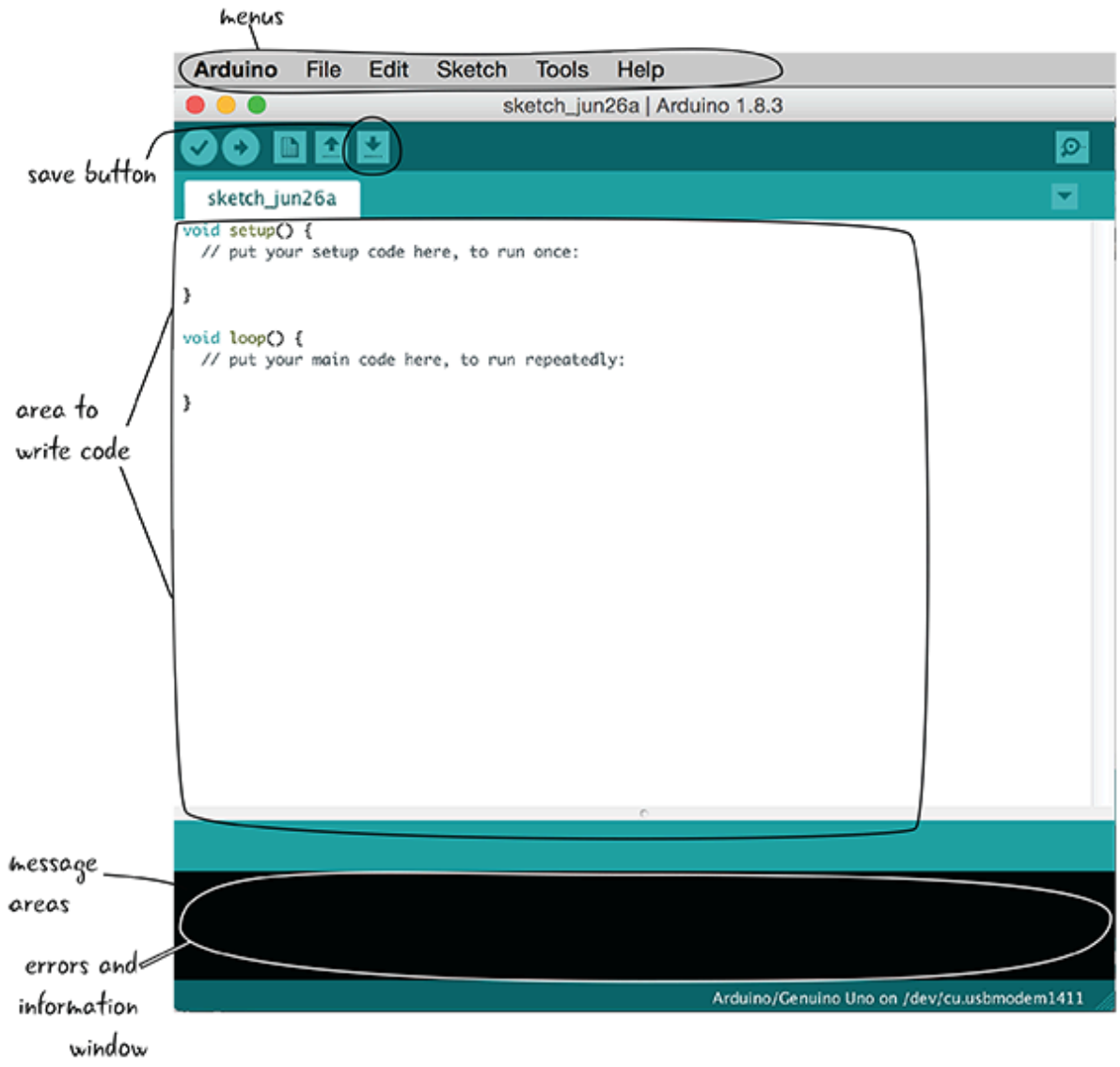


FIGURE 4-3: Arduino IDE

Now that you have an idea about what's in an IDE (and, specifically, in the Arduino IDE), you can download it and install it on your computer.

DOWNLOADING THE ARDUINO IDE: GETTING STARTED

The IDE you'll use to program your Arduino is free and available on the Arduino site. The installation procedure is slightly different for the Mac platform than for the Windows platform, so we'll walk you through the download and installation process for both of them.

> **Note**
>
> The URL again to download the IDE is arduino.cc/en/Main/Software.

IF YOU'RE USING A MAC

The download page will look something like Figure 4-4. Websites change frequently, as does software, so it may look different when you visit it. Click the link to download the Mac version of the software. Make sure you download the latest recommended version of the Arduino IDE for Mac.

FIGURE 4-4: Arduino IDE download for Mac

When you click the link, a zipped version of the Arduino IDE will start to download. It will be in the default download location on your computer, most likely the `Downloads` folder. When it has finished downloading, double-click the zipped file to unzip it. The unzipped file will be named `Arduino.app` and will look similar to Figure 4-5.

> **Note**
>
> If you don't see `.app`, don't worry—it means that your computer is set not to display file extensions.

FIGURE 4-5: Icons for Arduino app

Move the `Arduino.app` file into the `Applications` folder on your computer, as shown in Figure 4-6.

You have now downloaded and installed the Arduino IDE on your Mac.

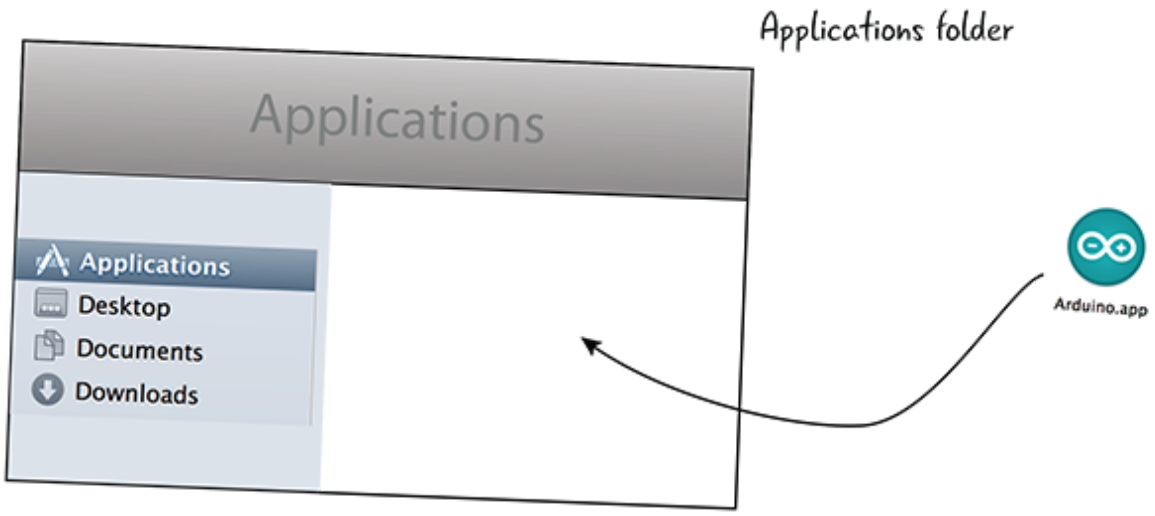


FIGURE 4-6: Drag the icon to your `Applications` folder.

IF YOU'RE USING A WINDOWS PC

Downloading and setting up the software for a Windows PC is very similar to the steps taken for a Mac computer, but there are a few additional minor steps you need to take in order to ensure the computer and the Arduino can communicate.

First you'll have to download the software. The URL is the same as that for the Mac download. Make sure you download the latest recommended version of the Arduino IDE for Windows, as shown in Figure 4-7.

> **Note**
>
> The URL again to download the IDE is arduino.cc/en/Main/Software.

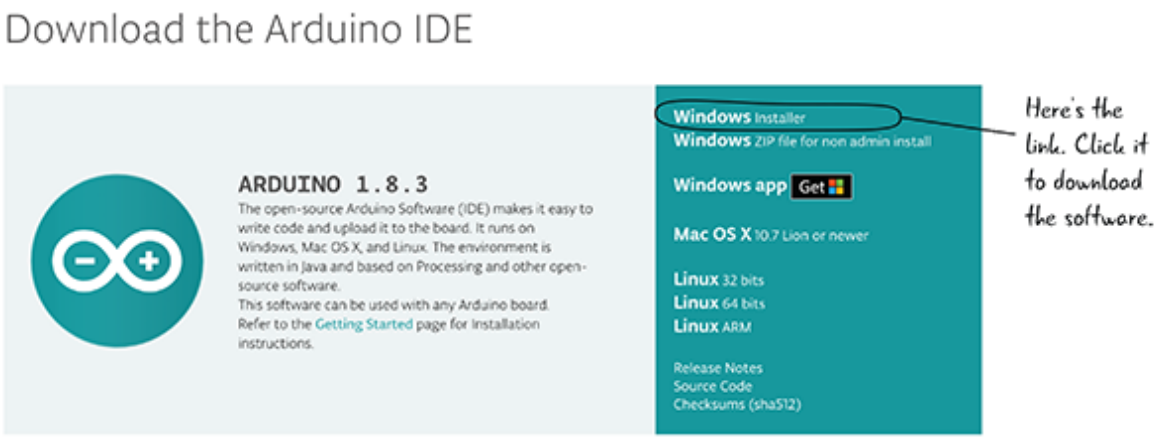


FIGURE 4-7: Arduino IDE download for Windows

We recommend you use the Windows Installer link. If you are sharing the computer—for example, you're using a computer at school or work where you are not the only user—you may need to download the version marked "non-admin install."

When it is finished downloading, there will be an EXE file named with the Arduino version in your default download location, generally the `Downloads` folder. Double-click on this file to start the installation process.

The first dialog box asks you to agree to the Arduino License Agreement (Figure 4-8). Clicking "I Agree" will take you to the next step of the installation.

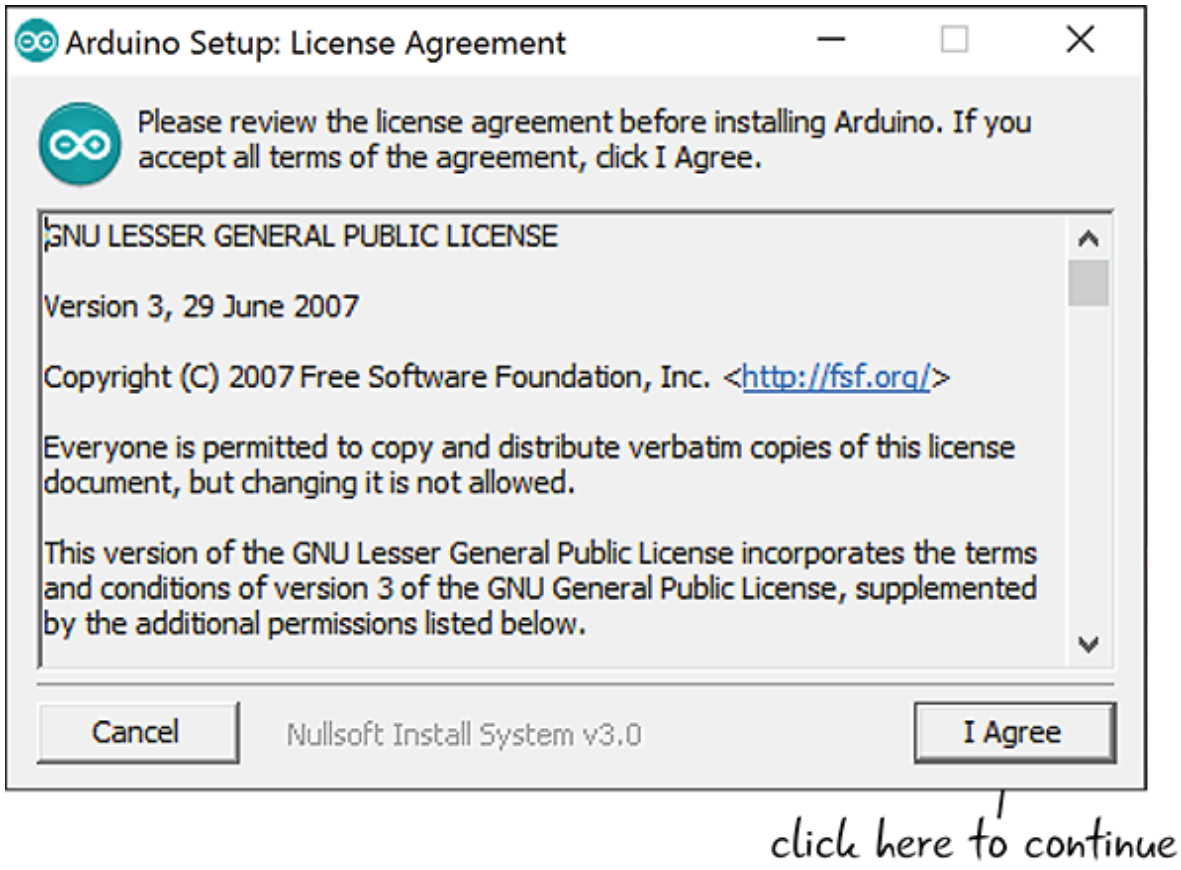


FIGURE 4-8: Arduino license agreement

With the Arduino Setup Installation Options, make sure that the Install USB Driver and the Associate .ino Files boxes are checked (Figure 4-9). Create Start Menu Shortcut and Create Desktop Shortcut are optional but will help you navigate to the Arduino IDE in the future.

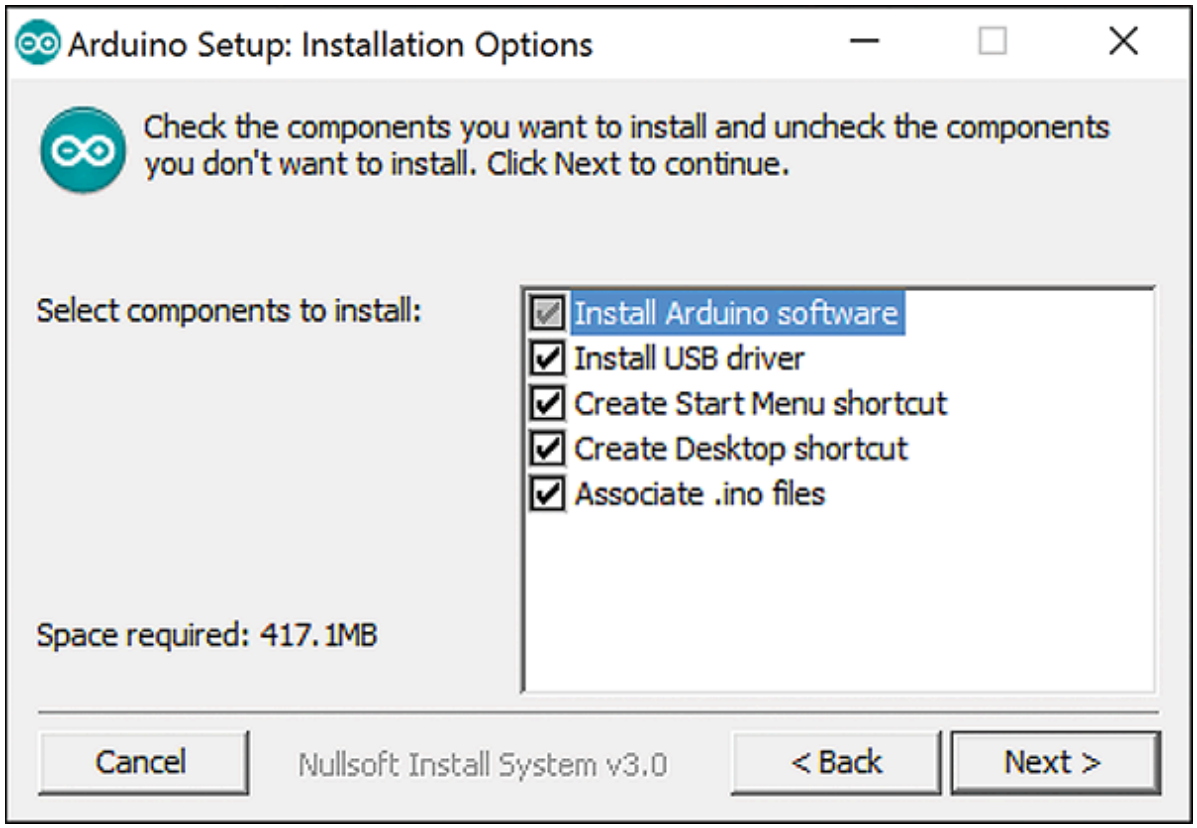


FIGURE 4-9: Installation options

Depending on your settings and your version of Windows, you may get a Windows security pop-up box asking about the USB Driver installation. Click Install whenever a security dialog box pops up to allow the Arduino IDE to be installed completely (Figure 4-10).

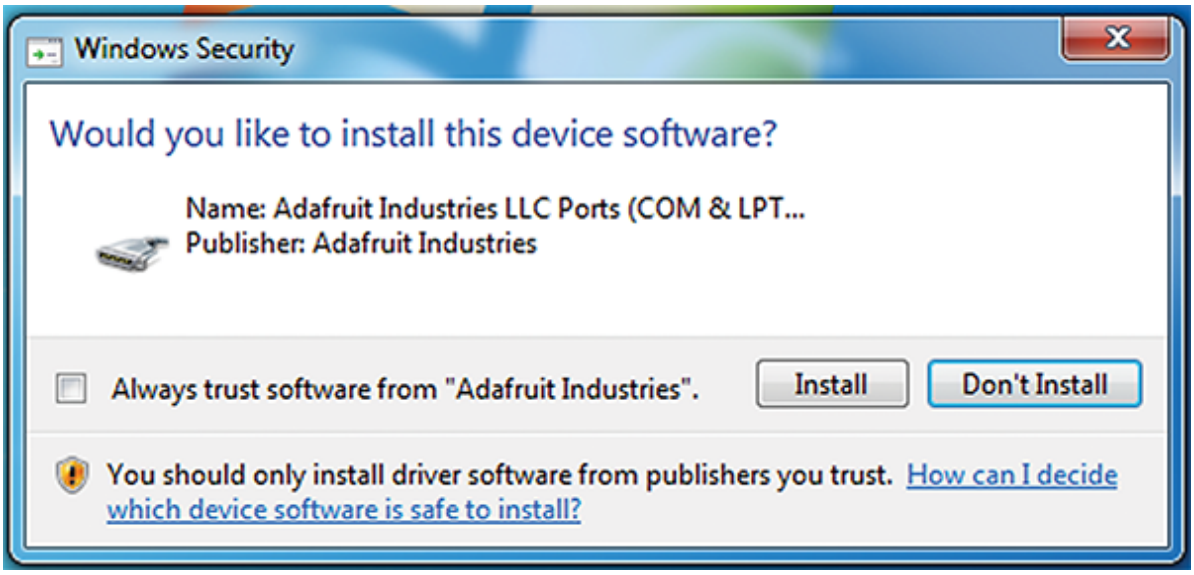


FIGURE 4-10: Security dialog box

That's it! Now your Arduino IDE is ready to run on your Windows PC.

CONNECT YOUR ARDUINO TO YOUR COMPUTER

You've installed the Arduino IDE, so now it's time to connect your Arduino to your computer so you can program it.

Plug your USB cord into the Arduino, and plug the other end of the USB into your computer, as in Figure 4-11.

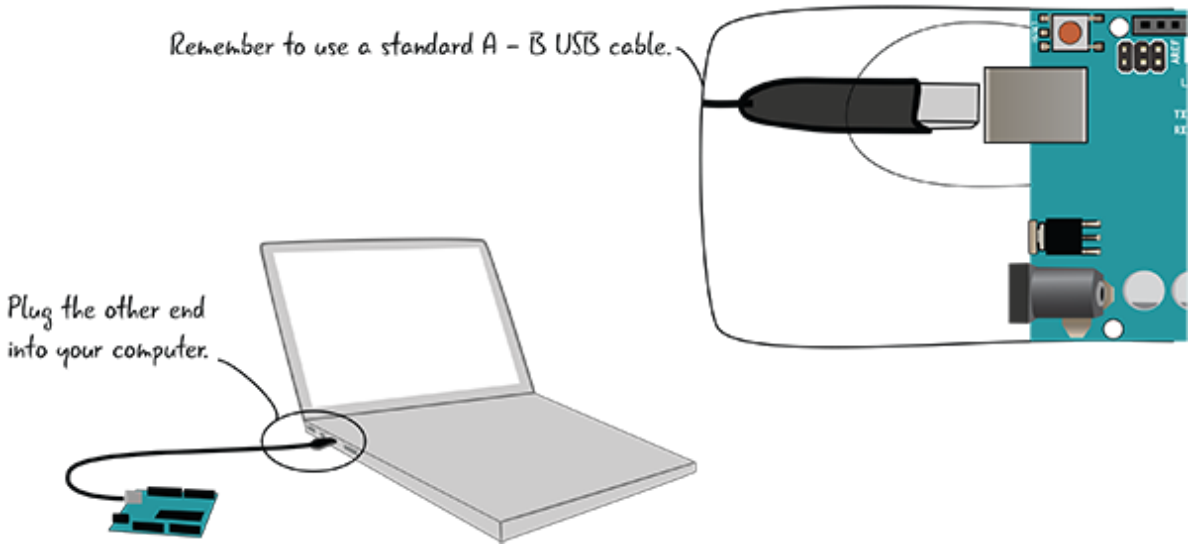


FIGURE 4-11: Attach your Arduino to your computer.

The LED marked ON should light up, and if your Arduino is brand-new out of the box, the light near Pin 13 should be blinking, just like when you tested plugging your Arduino in for Chapter 2 (Figure 4-12).

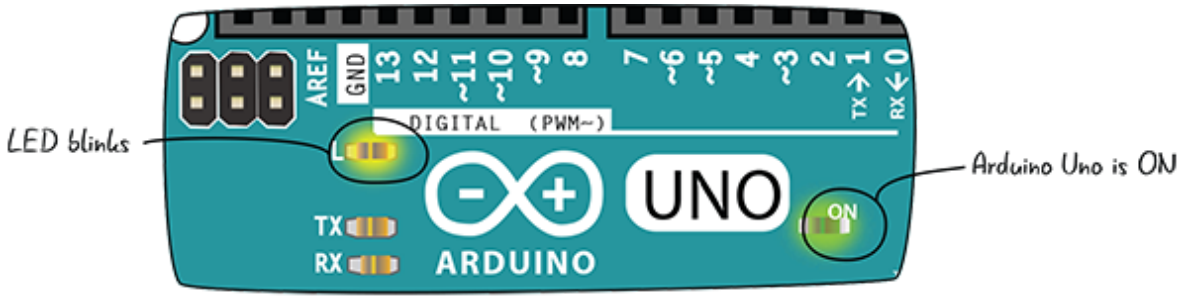


FIGURE 4-12: Indicator LEDs

THE ARDUINO IDE: WHAT'S IN THE INTERFACE?

Let's take a look at the Arduino IDE in Figure 4-13 now that you've launched it.

The Arduino IDE allows you to check whether your Arduino is connected to the computer, check your code for errors, upload any code you write to control your Arduino, and has a few other helpful options for understanding how the Arduino is behaving. We'll look at

all of the features in much more detail before you start to write code for your Arduino.

A program we write in the code editor for the Arduino is called a *sketch*. When you launch the software for the first time, you'll see the bare bones of a sketch. We'll explain how the code that's there is used as you start to program your Arduino.

> **Note**
>
> A *sketch* is the name for a program you write for the Arduino.

FIGURE 4-13: Basics of Arduino IDE

Warning

One peculiarity of the Arduino IDE is that if you close all of the sketch windows, the IDE will try to close. It will ask you to save a sketch if you made any changes, but otherwise it will close.

You'll have to configure some settings before you start programming. Let's look at them now.

CONFIGURING THE IDE

Two important settings need to be configured in the Arduino IDE so your computer can communicate with your Arduino Uno. You need to specify which version of the Arduino hardware, or board, you are using, and which connection or port you'll use for communication between the Arduino and your computer. These settings will be the same as long as you're using the same Arduino Uno. (The settings will be different if you're using another Arduino. We're using the same Arduino for all of the projects in this book.)

Specify the Arduino Hardware Version

You saw in Chapter 1 that there are many different versions of the Arduino. To program yours, you must indicate in the software which version of the Arduino board you're using.

To do this, go to the Tools menu and select Board, as shown in Figure 4-14. From the flyout menu, select Arduino Uno/Genuino. Once this is set, you'll have to set a port through which your Arduino will communicate with your computer.

FIGURE 4-14: Selecting the Arduino board

Specify Which Port You're Using

There is a port on the Arduino that communicates with a port on your computer when the two are connected by a USB cable. Think of the port as the channel through which the two devices speak to each other. Right now, you need to set up the Arduino IDE so the correct port on your computer communicates with your Arduino.

Selecting the right port looks slightly different on a Mac and a Windows computer. We're going to look at screenshots for both of them. Remember, you're setting up your computer to talk with your Arduino Uno, since that is the version of Arduino that we're using for the projects in this book. Let's look at Mac first; if you have a Windows PC you can skip ahead to the next section.

Note

A *port* is a channel of communication that connects your Arduino and the computer.

Mac Port Selection

To set the correct port for your computer to communicate with the Arduino, go to the Tools menu and select Port, as shown in Figure 4-15.

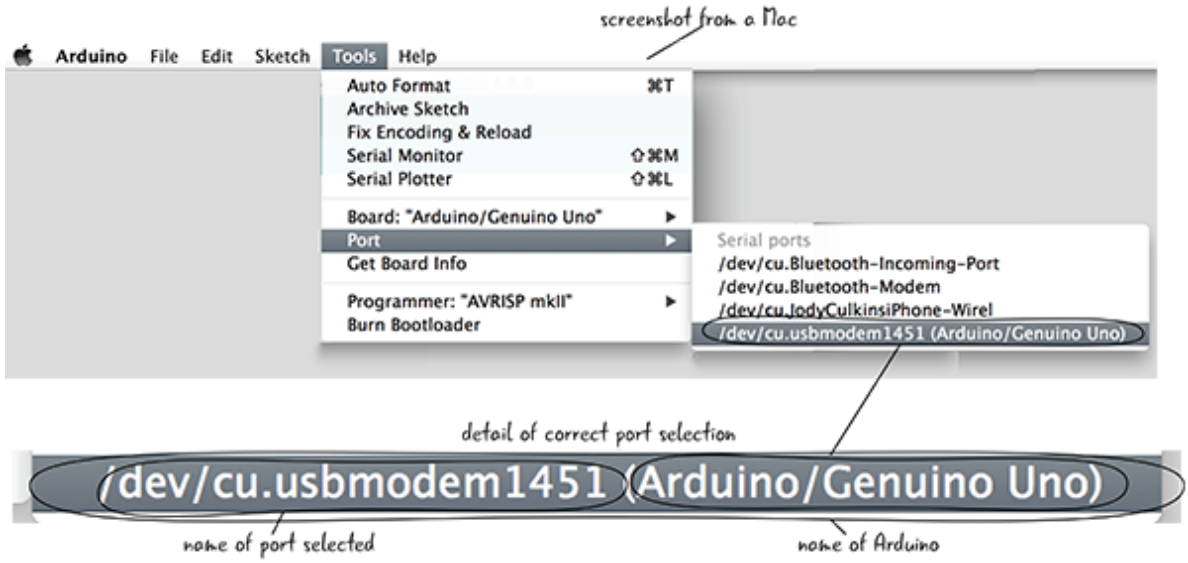


FIGURE 4-15: Selecting the correct port

On a Mac, select the port whose description includes *dev* and *cu* and that is labeled Arduino/Genuino Uno. Dev is a prefix added by the Mac, cu is short for call-up, and Arduino Uno is the version of Arduino hardware you're using. In our earlier example, the number at the end of that menu item is 1451; on your screen, this will be different from this example, and it might be different each time you connect your Arduino. In some versions of the software or operating system, you may see *tty* rather than cu in the lists of ports. That

should work as well; what is important is that you see Arduino/Genuino Uno in the port description.

Nothing bad will happen if you select the wrong port, but the Arduino and your computer won't know how to talk to each other. If it seems that the Arduino and your computer aren't communicating, take another look at your list of ports and make sure you've selected the right one.

Windows Port Selection

Let's look at the port selection on a Windows machine (Figure 4-16). On a PC, the port names will all start with COM. You want to go to the Tools menu, select Port, and then select whichever COM number matches up with the Arduino Uno/Genuino label under Serial Ports. It will be something like COM3 (Arduino Uno/Genuino).

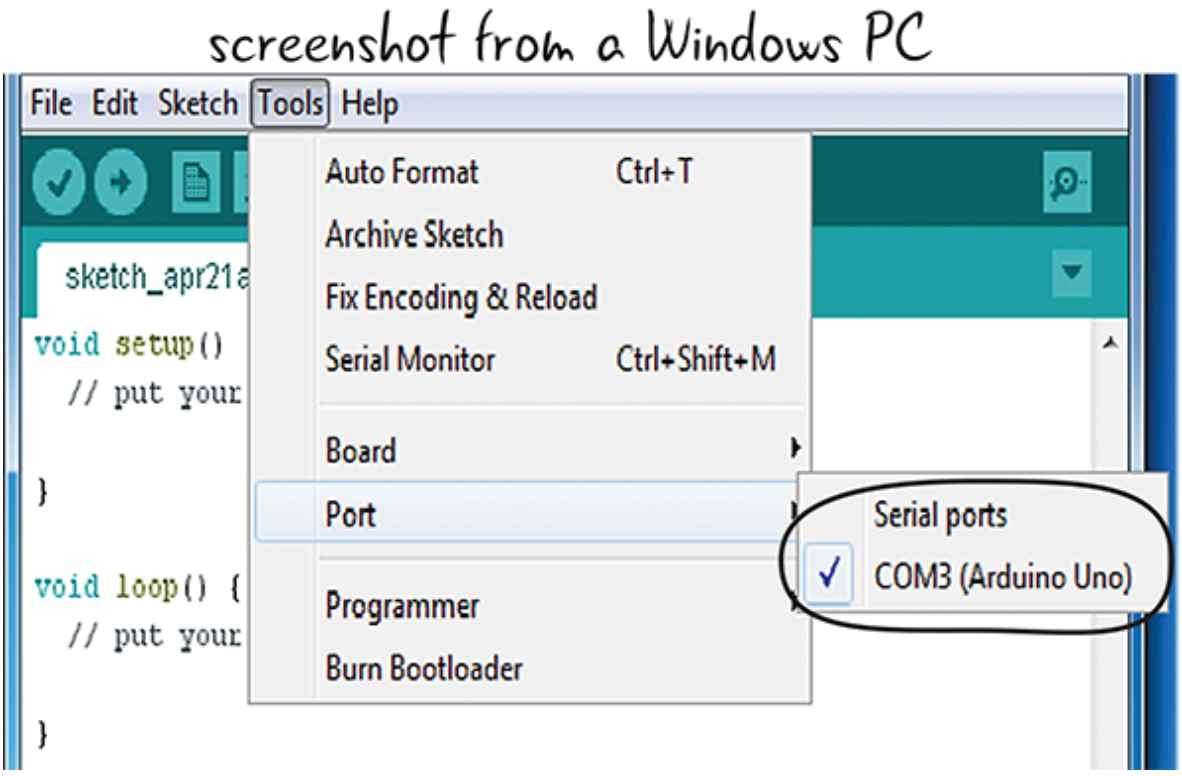


FIGURE 4-16: Selecting the correct port

QUESTIONS?

Q: Will I always choose the port that says Arduino Uno/Genuino?

A: Not necessarily. That is the version of the Arduino board we are using in this book, so all of the projects in the book use the Arduino Uno, but you may want to use other versions of Arduino as you build your own projects later on.

Q: Sometimes there are other ports listed in the dropdown. What are they?

A: Those are other ports that use different means for your computer to communicate with other devices. Don't worry about them—we won't be using them.

Q: What if I don't have my computer hooked up to the Arduino? Will I see the port to connect to my Arduino then?

A: No. In order to see the correct port, you must have your Arduino and computer connected with a USB cable.

Now that you've set the port and the correct Arduino board, let's take a closer look at the Arduino IDE used to create your code.

UNDERSTANDING THE CODE WINDOW

We've heard about the parts of the Arduino IDE; now let's take a look at them more closely in Figure 4-17.

As in most software, there are menus that allow you to perform various actions, such as creating new files, saving them, and many more, at the top of the software interface. There are button icons that also allow you to quickly access some of the most often performed actions. Clicking the Verify button checks to make sure

there are no errors in your code. Clicking Upload transfers your code from your computer to your Arduino so it can run on your Arduino board. There is a window where you type your program, and message areas that give you information about that program. We'll explain more about messages as we work in the IDE; for now, know that they tell you if your code has errors, and also information like the amount of space it uses in the Arduino's memory.

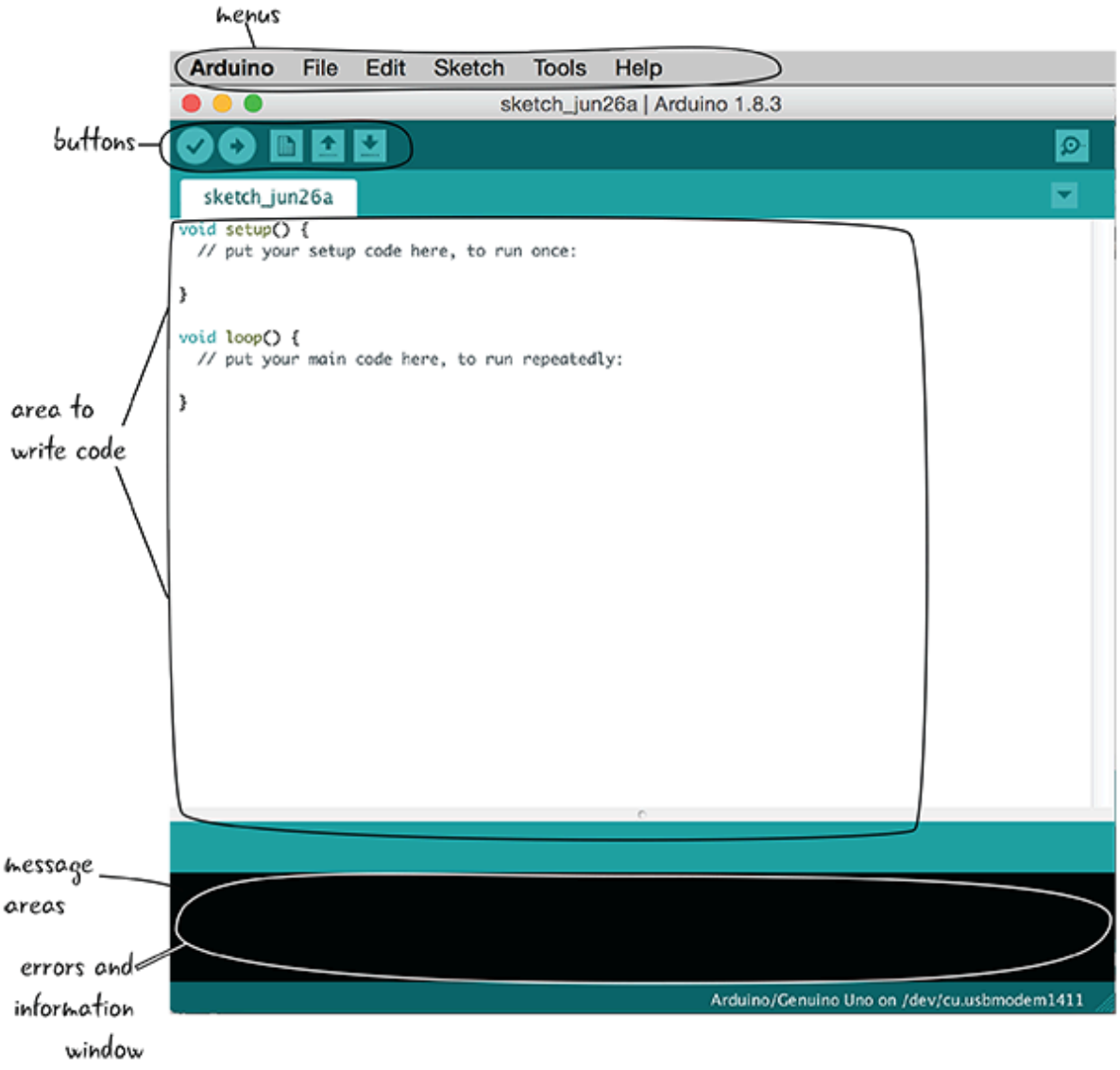


FIGURE 4-17: Arduino IDE annotated

Let's look a little closer at the buttons at the top of code editor in Figure 4-18.

These buttons allow you to quickly access the actions that you will perform most often with the code window. These actions include checking if your code has any errors (verifying), sending your code to the Arduino board (uploading), creating a new file, opening a file, and saving it.

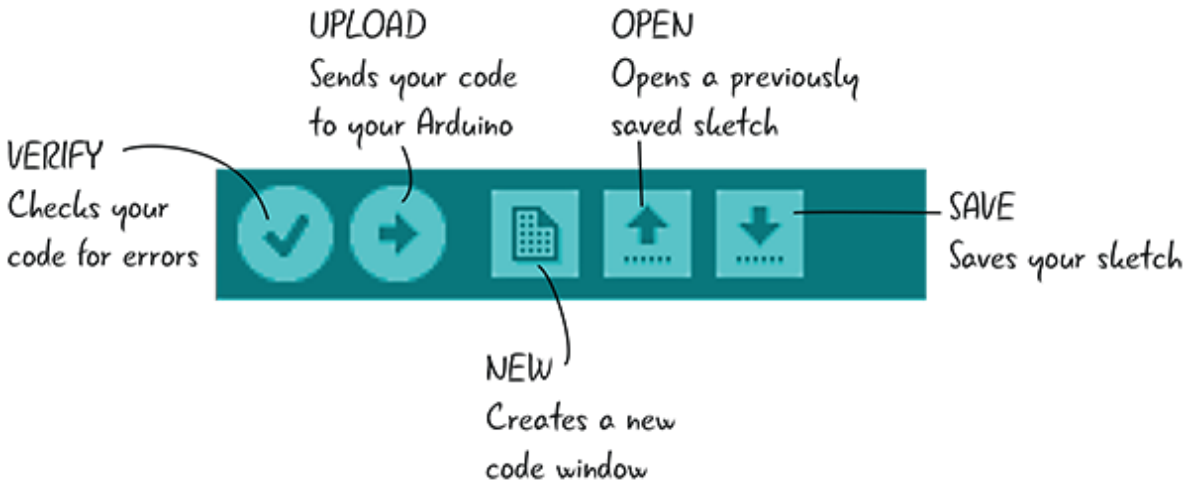


FIGURE 4-18: Buttons in the Arduino IDE

We'll use all of these buttons in just a moment, but first let's be clear about what writing a sketch actually means.

THE SKETCH: THE BASIC UNIT OF ARDUINO PROGRAMMING

You can think of an Arduino program, or sketch, as one full group of instructions to perform specific tasks. A sketch includes all of the code, or instructions, for that task or tasks. It's possible to have multiple, separate sketches open at once—just as a spreadsheet program can have more than one sheet open at a time. Let's take a closer look at what forms a sketch.

Every program you upload to your Arduino is considered a *sketch*. A sketch can be quite simple or extremely complex. It could turn a single LED on and off, or it could control 10 or more motors based on sensor input. Although each sketch corresponds with one task, that task could be made up of multiple parts. For example, your program may take measurements of the world (like light levels) and use them to trigger speakers and LEDs. All of that would go in one sketch.

The name of the sketch appears in a tab in the upper-left corner of the code editor. Figure 4-19 shows examples of Arduino sketches.

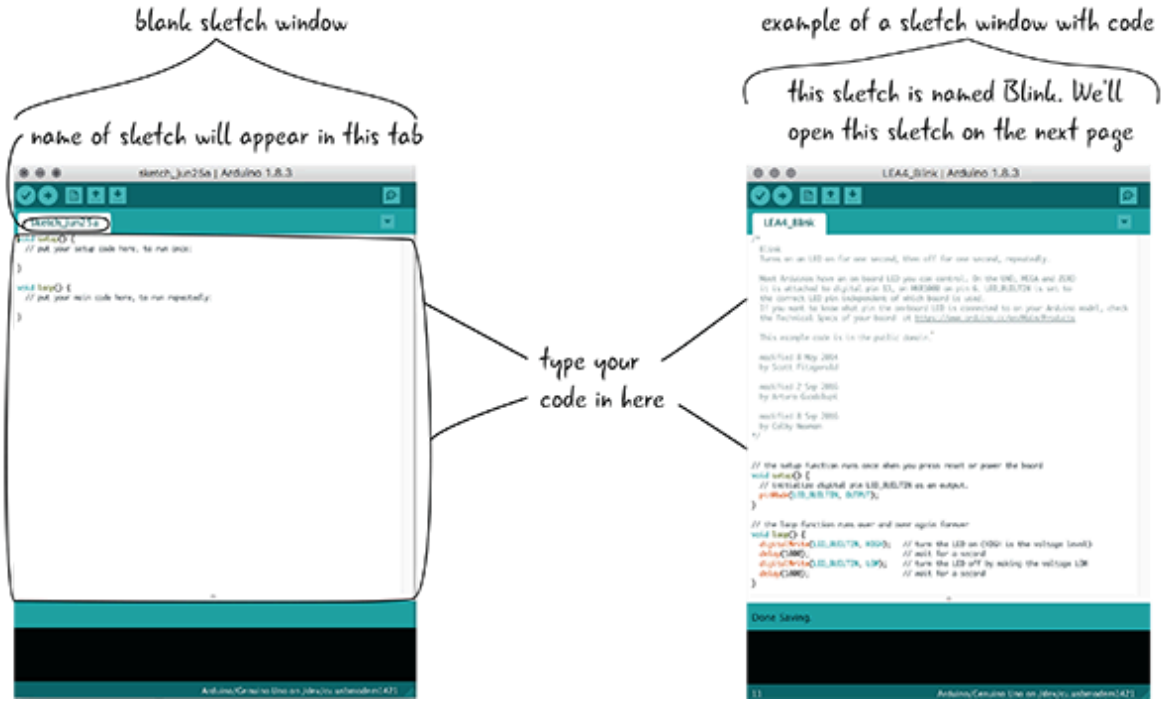


FIGURE 4-19: A blank sketch window and one with code written in it

OPENING AN EXAMPLE SKETCH

Before you start to write your own code, let's explore an example that is included in the Arduino IDE. The IDE has a lot of examples (sample code) that demonstrate many of the things that the Arduino can do built into it. You can load an example into the code window and upload it to your Arduino when it is attached to your computer.

First, open up the example sketch named Blink by selecting File > Examples > 01.Basics > Blink, as shown in Figure 4-20.

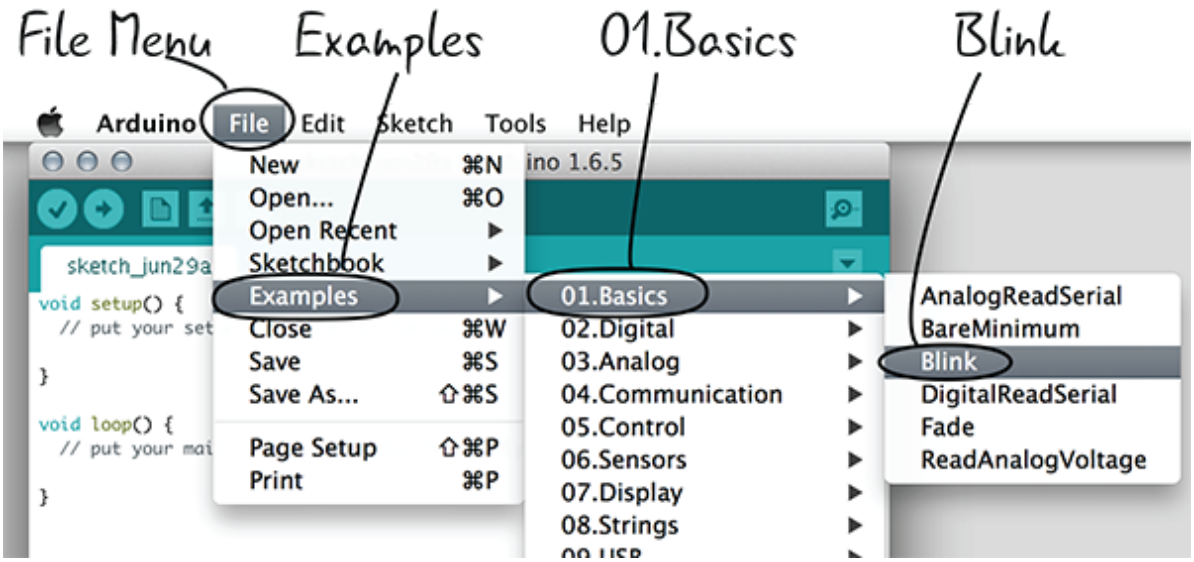


FIGURE 4-20: Opening the Blink sketch

SAVING YOUR SKETCH

By default, your Arduino sketches will be saved inside the Arduino folder within your computer's `Documents` folder. It is a good idea to continue to save in this space since it makes it easier to return to the files. Arduino also keeps track of past files saved inside this folder in the Sketchbook dropdown in the File menu.

Even though you're using code from an example, it is best to save it now with a different name so that you can always return to the original unadjusted example code later. That way, when you make changes and save your sketch you'll know you haven't saved over the Blink example sketch accidentally. Save your sketch as LEA4_Blink so that you'll be able to find your changes later.

SAVE EARLY, SAVE OFTEN!

Get in the habit of saving your files. Just like you wouldn't want to lose work from a paper or another project, saving early and often can help save frustration if for some reason your computer closes the

Arduino IDE (losing power, momentary hiccup, etc.). Although the odds of this happening are low, the one time it does you'll be glad you don't have to repeat all the work you did because you saved your project and don't have to worry about it.

> **Tip**
>
> Keep saving your sketch files as you are working.

UPLOADING A SKETCH TO THE ARDUINO

Now that you've saved the example sketch with a new name, it's time to upload it to the Arduino. Before you upload it, let's check it for errors. Even though you're using the code that's built into the IDE, get in the habit of always verifying your code before you upload it.

There are two buttons we talked about earlier that you need to keep in mind when you're ready to upload your code: Verify and Upload. We've highlighted both of these buttons in Figure 4-21.

FIGURE 4-21: Verify and Upload buttons in the Arduino IDE

Step 1: Verify Your Sketch

Verifying ensures that your code is set up correctly. Click the Verify button to make sure there are no errors (Figure 4-22). Unless you made changes to the LEA4_Blink sketch before you saved it, everything will work fine.

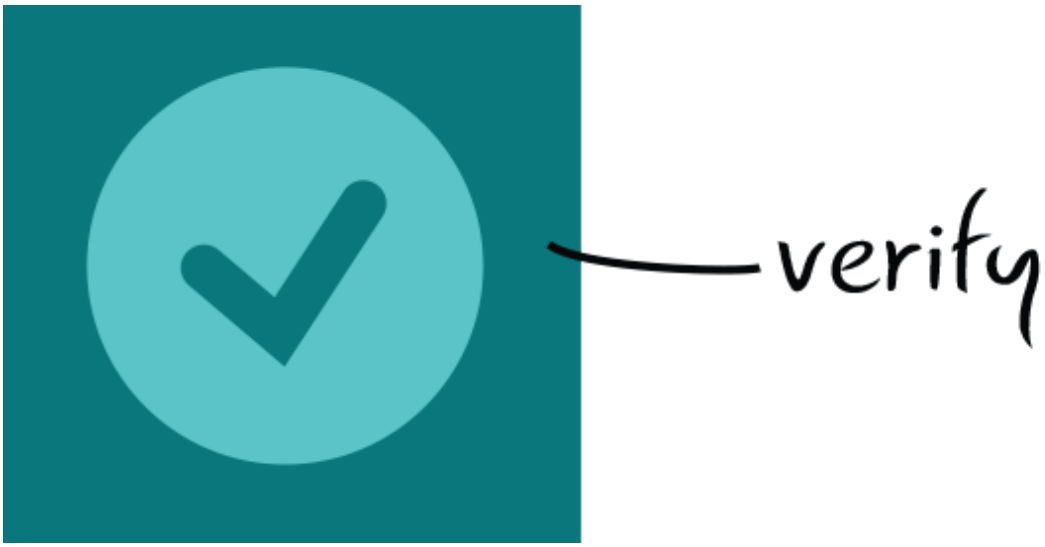


FIGURE 4-22: The Verify button

The message window at the bottom of the IDE shown in Figure 4-23 will display "Done compiling" and show no errors.

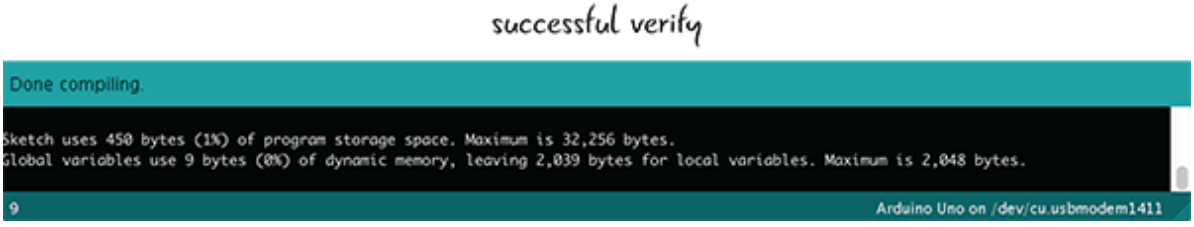


FIGURE 4-23: The message window

When you verify your code, you will get a message that notifies you that something is wrong if there are any errors in your sketch. The Arduino IDE only knows about programming errors, not mistakes you might have made in setting up your circuit with the Arduino. (We'll cover those types of errors as we progress through the book.) When we type text into the Arduino IDE window, the code looks like something that humans can read, but the Arduino doesn't understand how to interpret it. Your computer temporarily converts the code into a language that the Arduino understands when you click Verify to check for these errors.

Step 2: Upload Your Sketch

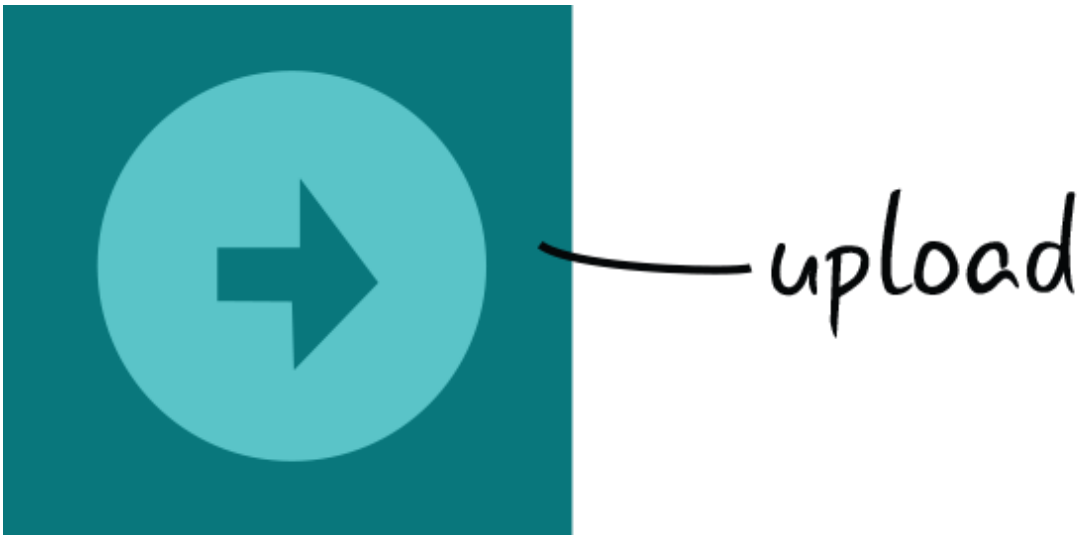


FIGURE 4-24: The Upload button

When you click Upload (Figure 4-24), your computer converts the code into a language that the Arduino understands and then immediately begins sending this program over the USB cord to your Arduino.

Uploading Continued: Status Bar and Message Window

Once you click the Upload button, the Arduino IDE window will give you a status bar indicating how much progress the upload has made and a message window with information such as the size of the sketch. That progress bar and message window looks something like Figure 4-25.

Once the file has been sent to your Arduino, the message window will say "Done uploading."

That's it! Now your code from the IDE window is running on the Arduino.

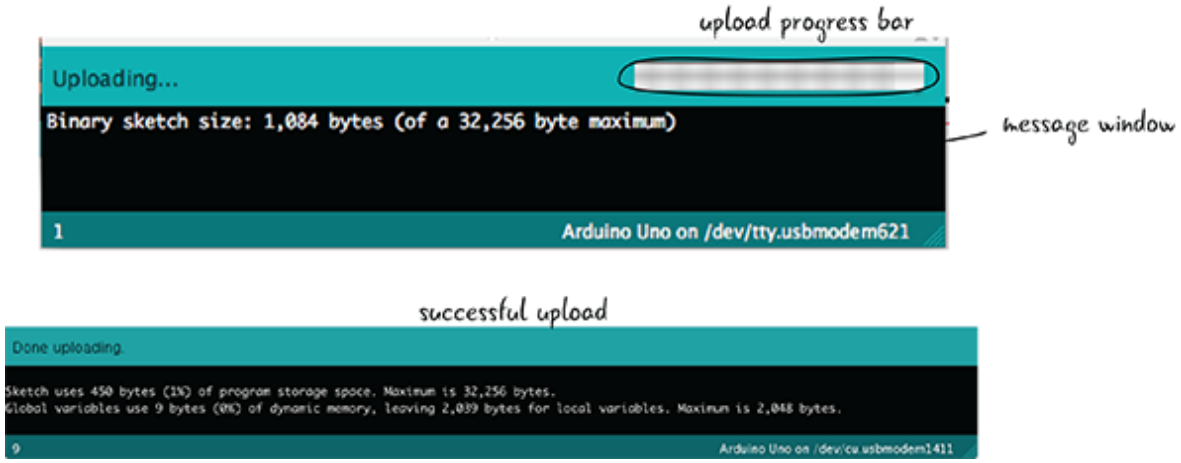


FIGURE 4-25: Upload progress bar

Run the LEA4_Blink Sketch

Now that you've uploaded your sketch to the Arduino, as long as the Arduino has power from the computer through the USB cable, it will keep running. The code that you've uploaded to the Arduino contains the instructions that tell the Arduino to blink the light over and over. The LED near Pin 13 will turn on and stay on for one second, then turn off for one second, over and over again. This is illustrated in Figure 4-26. We'll look at the code in detail shortly and see exactly how it works.

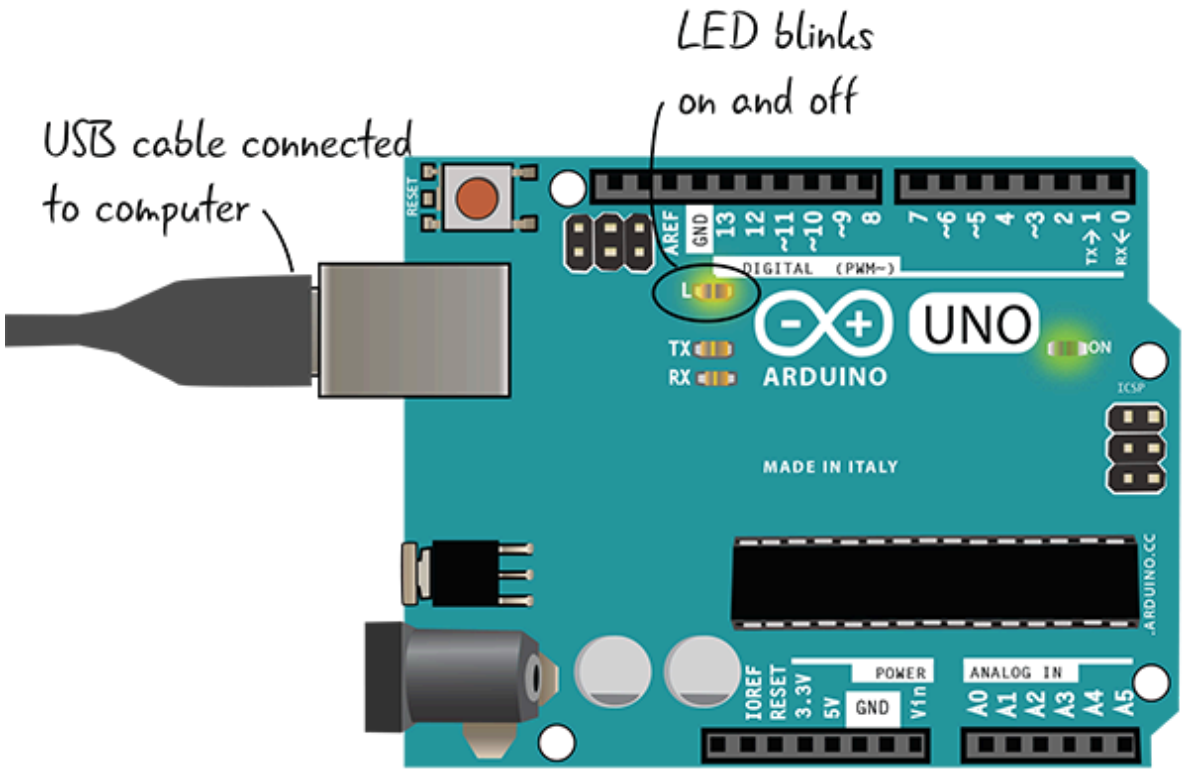


FIGURE 4-26: The LED blinks.

If your LEA4_Blink sketch is not running, you can turn again to the methodical process used to discover what issue is preventing your code from working. You've seen this before with our electronics, and it is known as *debugging*.

Note

Debugging is the name of the process used for solving issues with the circuit and with the code in your Arduino projects.

DEBUGGING: WHAT TO DO IF THE LED ISN'T BLINKING

If the upload was successful and your LED is blinking, there isn't anything to fix. But what if the LED didn't light up? Just as you used debugging to search out issues in your circuit, you'll debug your code throughout the book, methodically looking for problems that prevent your code from functioning properly. You'll also look for problems with how the Arduino hardware is set up. If you had any issues with your LEA4_Blink sketch make sure that:

Your USB cord is tightly plugged into both your computer and your Arduino (Figure 4-27).

You have selected the right board type and serial port from the menus (Figure 4-28).

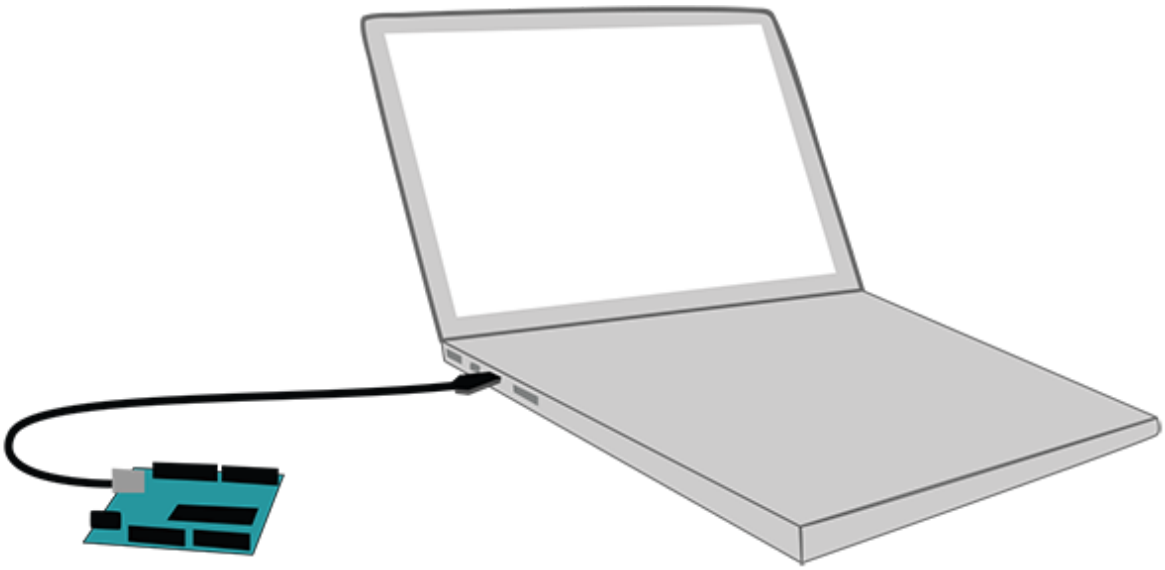


FIGURE 4-27: Make sure your computer is firmly attached to your Arduino via the USB A-B cable.

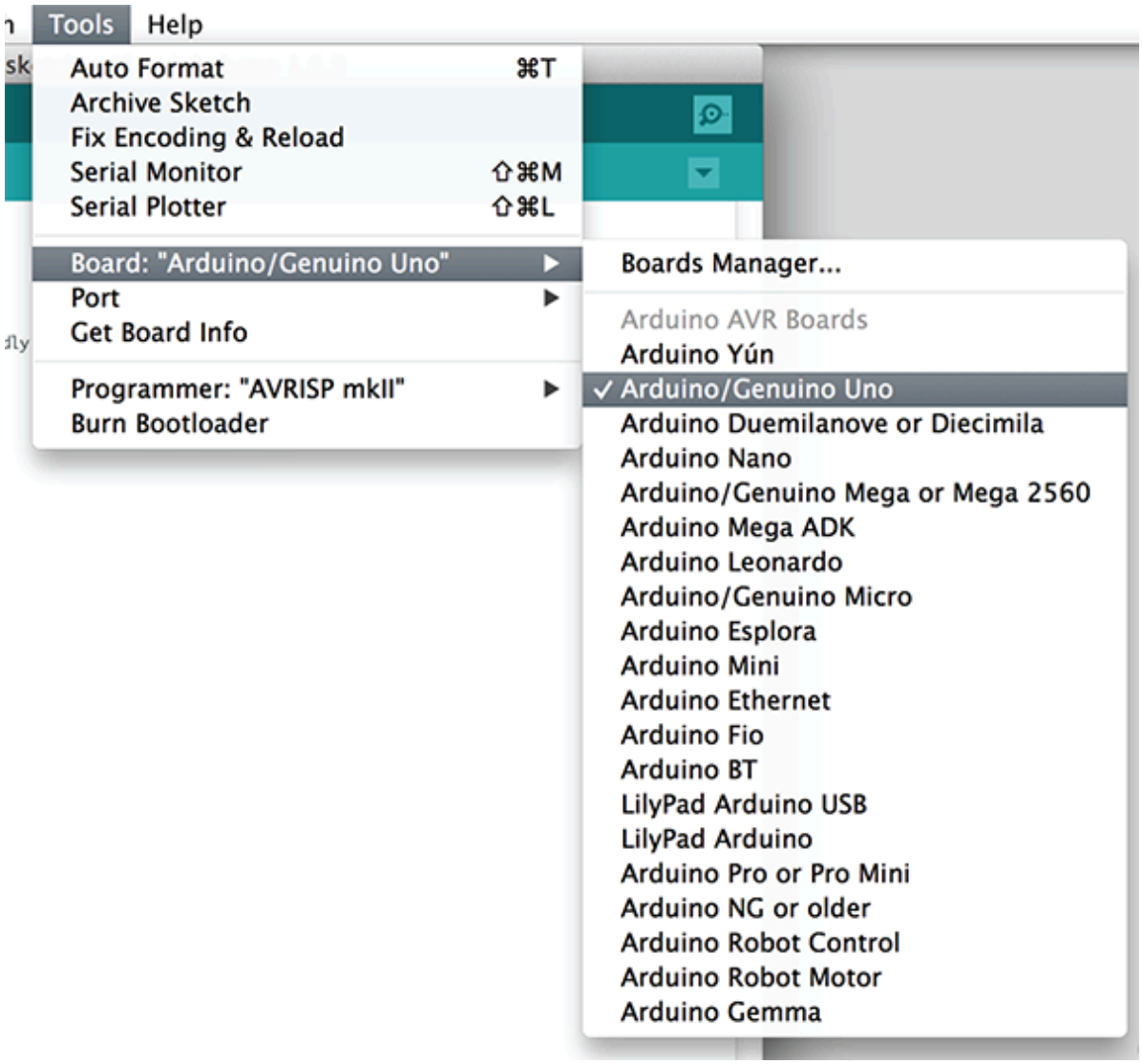


FIGURE 4-28: Make sure you have selected the correct board.

If your Arduino seems not to be responding, you can always push the Reset button before uploading, as shown in Figure 4-29. The Reset button will turn off your Arduino for a moment before turning it back on.

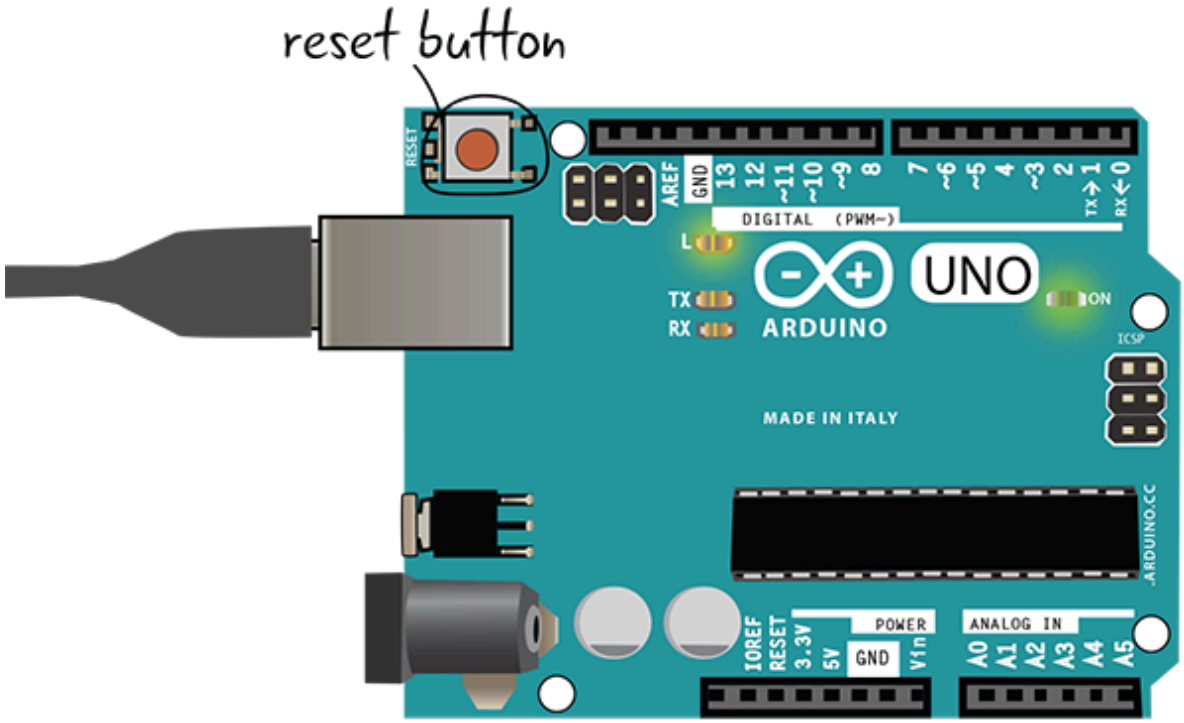


FIGURE 4-29: The Reset button on the Arduino

You can also try switching your USB port or restarting your computer if none of the above solutions work for you. We'll cover all sorts of code debugging tricks throughout the book, but these few basic tips concerning the Arduino can save you a lot of headaches later.

The LEA4_Blink sketch will run as long as the Arduino has power, but how does it actually work?

LEA4_BLINK SKETCH: AN OVERVIEW

Figure 4-30 shows a screenshot of the LEA4_Blink sketch. This is a quick overview of the parts of the sketch; after we look at it we'll go over every part of it in detail.

FIGURE 4-30: LEA4_Blink sketch first look

Comments are notes to the programmer in the text that are not part of your program. In an Arduino sketch, `setup()` is where you put the parts of your program that happen only once, and `loop()` is where you put what you want to happen over and over again.

In this sketch, all of the code in `setup()` and `loop()` is written in the Arduino programming language. If you look at the code in the Arduino IDE, you will see that parts of the code are different colors; some are orange, some blue, some black. These colors represent some of the different roles of the code. It's not important to memorize or know these colors; they are just there to help you visually separate the purpose of the various parts.

> **Note**
>
> In the LEA4_Blink sketch, all of the code in `setup()` and `loop()` is defined by the Arduino programming language.

We'll look at all the parts of a sketch in detail shortly, but first let's look at the comments section at the top of the code.

COMMENTS: LETTING OTHERS KNOW WHAT YOU'RE THINKING

Comments in code are used to write notes to anyone who might read your code. This might be you when you return to a sketch you created earlier, or others with whom you are sharing your code. Comments have no effect whatsoever on how the computer interprets the text and are only there to remind you or clue someone else in on what you intended to do, or how the program functions as a whole. We'll use comments throughout our code examples to help explain sections. It is a good habit to get into writing comments to yourself so that you can return to a sketch later and remember what is going on.

The first part of the LEA4_Blink sketch includes a comment about how the file works. This comment is long, with lots of information, but comments are sometimes short, just a word or two. As you can see from this example, comments sometimes have information about the author of the code and the date. In this case they also tell us that the code is in the public domain.

In the Arduino language, as in many other popular languages, there are a couple of ways to indicate comments. Multi-statement comments start with `/*` and end with `*/`, which allows for entire blocks of code to be commented out. Single-statement comments start with `//` and end when you hit Enter to create a new statement. Sometimes the single statement comments are at the end of a

statement of Arduino code. Anything written after the double slash (//) will be ignored until the next statement.

As you can see in Figure 4-31, the top section of the LEA4_Blink sketch shows the comments at the beginning of the sketch.

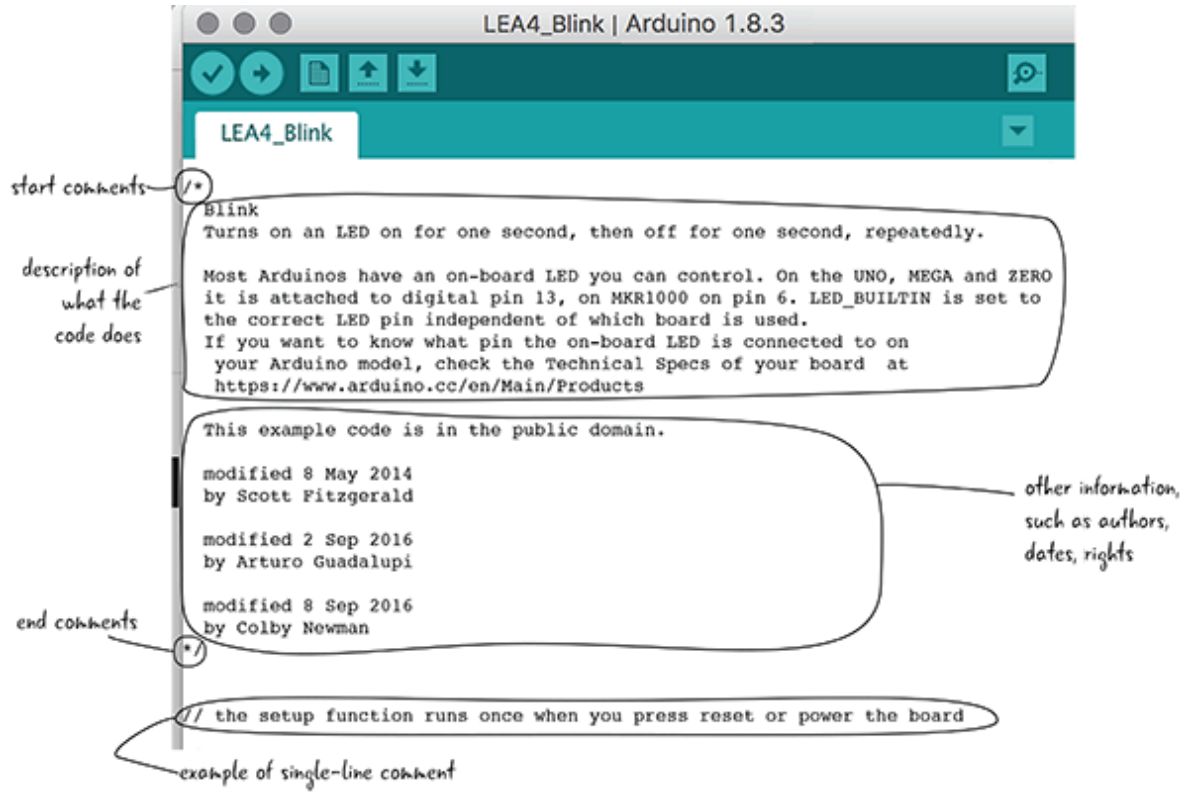


FIGURE 4-31: Comments at the beginning of the LEA4_Blink sketch

Note

/* and */ denote the beginning and end of a block comment.

// indicates single-line comments.

SETUP() AND LOOP(): THE GUTS OF YOUR CODE

Comments, though important, are not instructions to the Arduino. In an Arduino sketch, there are two basic sections: the `setup()` function and the `loop()` function. The diagram in Figure 4-32 shows how `setup()` and `loop()` work: `setup()` happens once, followed by `loop()`, which repeats over and over.

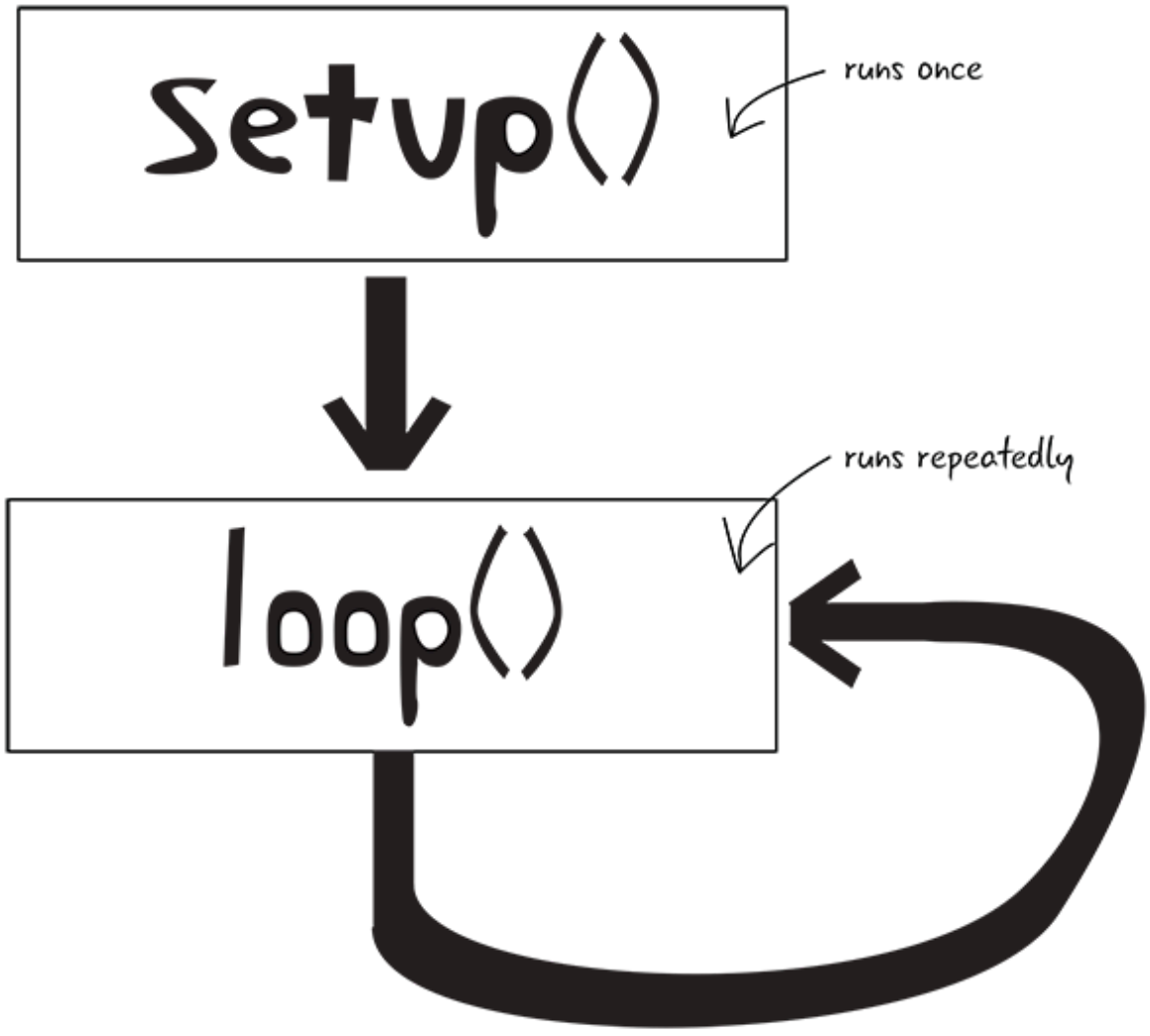


FIGURE 4-32: `setup()` and `loop()` diagram

`setup()` is the name of a function that is included in every Arduino sketch. What's a function? Just think of it as a way of organizing code or instructions to the computer.

> **Note**
>
> A *function* is a way of grouping statements of code or blocks of instructions to the computer.

Generally speaking, anything that you want to happen only once in your sketch belongs in the `setup()` function. `setup()` is run exactly once every time your Arduino is reset.

SETUP() AND *LOOP()* CONNECTED

We are going to take a look at the rest of the Arduino code for the LEA4_Blink sketch, but first let's look at a couple of example projects. We'll use these example projects to help you gain an understanding of the difference between the `setup()` and `loop()` sections of the code.

EXAMPLES: HOW DO *SETUP()* AND *LOOP()* APPLY TO THE PROJECTS?

Example 1

Later on in this chapter, you will be building an SOS signal light that will flash an LED on and off in an SOS pattern continually, with the timing controlled by the programming.

`setup()` is where you will set the pin that controls the LED to an output, telling the Arduino which pin will control the LED.

`loop()` is where you will put the code that controls the timing, turning the LED on and off continually.

Example 2

Let's say you want to build a digital music box that plays different sounds depending on what button you press, and you want to include a volume control knob.

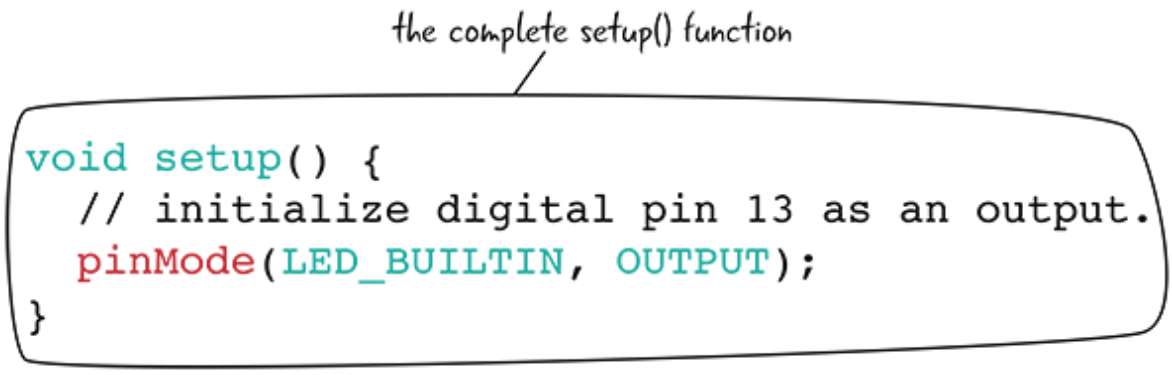
`setup()` will be used to assign different buttons to each of the sounds and determine which pin responds to the knob.

`loop()` will be focused on responding to button presses and playing each sound when the corresponding button is triggered. The `loop()` function will also look for changes in the knob, which will change the volume.

You've seen examples of how `setup()` and `loop()` work; now let's see what the `setup()` function looks like from the LEA4_Blink sketch you just uploaded and have running on your Arduino.

SETUP(): SETTING INITIAL CONDITIONS

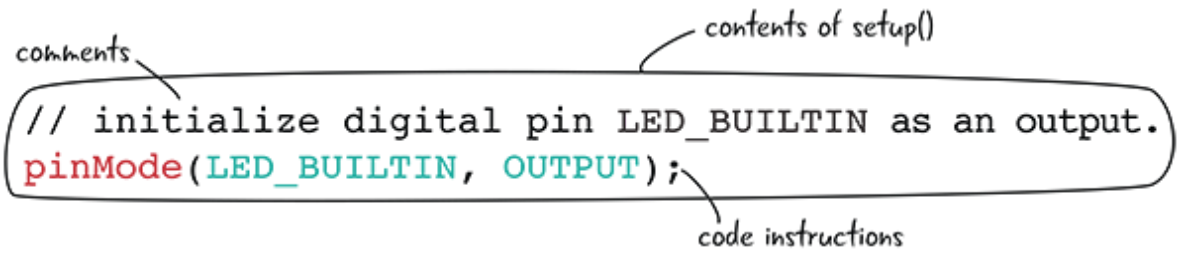
We've discussed comments, and you've seen that `setup()` runs once at the beginning, whenever you turn on or reset the Arduino. Let's now take an in-depth look at the `setup()` function in the LEA4_Blink example sketch.



`setup()` has some parentheses attached to it; we'll explain why they are needed and what they do later. After the parentheses is an opening curly brace, `{`, which is very important. Curly braces denote a block of code and mark off the instructions that will happen when the code is run. In this case, whenever `setup()` is run, all of the instructions will execute one by one. When you are done with the code block, be sure to include a closing brace, `}`, to tell the Arduino that you are done talking about that section of code instructions.

> **Note**
>
> *Curly braces* denote when we begin and end a block of code.

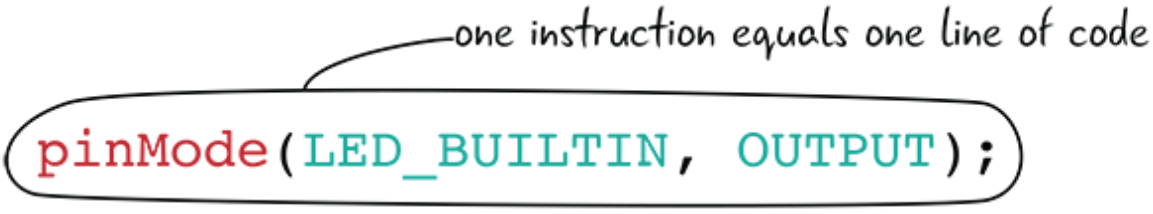
Let's take a look at what code instructions happen when `setup()` is run. For the LEA4_Blink sketch code, there is only one `setup()` code instruction and one single-line comment.



The first line looks familiar. It starts with two forward slashes, which means that it is a comment. In this case, the comment is telling us the purpose of the second line is to "initialize digital pin `LED_BUILTIN` as an output." We don't yet understand what this means, but we do now know that `pinMode(LED_BUILTIN, OUTPUT);` sets the built-in LED to be an output. Let's take a look at the code instruction line without the comment.

Note

A line of code is defined by one and only one instruction ending with a semicolon.



Semicolons in the code serve the same purpose as periods in English; they denote that you have reached the end of the line. This

keeps the Arduino from misinterpreting your instructions, because it knows you meant to end the line as soon as it sees a semicolon. If you omit the semicolon, you will generate an error in the Arduino IDE and your code won't upload to your Arduino.

```
pinMode(LED_BUILTIN, OUTPUT);
```
semicolon

> **Note**
>
> Semicolons end statements of code, like periods end sentences.

Next, let's look at the end of the line. `pinMode()` is followed by a set of parentheses, which contain the text `LED_BUILTIN`, a comma, and the word `OUTPUT` all in capital letters. `pinMode()` is a function that sets our pins to behave in a particular way.

> **Note**
>
> When we want to use a function or instruction like `pinMode()`, we say that we are "calling" the function.

```
pinMode(LED_BUILTIN, OUTPUT);
```
set pin mode

When you call `pinMode()`, you instruct the Arduino to set the pin with the number you type to act as either an input or an output. Instead of seeing a pin number there, you see `LED_BUILTIN`—which is there instead of the pin number, because your Arduino Uno knows

that `LED_BUILTIN` means Pin 13. So, 13 is the number of the pin you'll set to `OUTPUT`. You haven't wired anything to the Arduino yet, but the Arduino already has a tiny LED wired permanently as part of the board attached to Pin 13, which is where the word `LED_BUILTIN` originates. You are setting the mode for Pin 13, telling the Arduino that you plan to use Pin 13 as an output.

> **Note**
>
> `LED_BUILTIN` is connected to Pin 13 on the Arduino. Both `pinMode(13, OUTPUT)` and `pinMode(LED_BUILTIN, OUTPUT)` have the same outcome.

`OUTPUT` means that you want to control whether the pin is off or on. `OUTPUT` allows you to set the pin dynamically and change its state as your sketch continues.

SETUP(): IT HAPPENS ONCE

To recap, in our LEA4_Blink sketch `setup()` tells the Arduino to treat Digital Pin 13 like an output. The Arduino is good at remembering instructions that you tell it about pins, so you need to tell it only once. As long as this sketch is still running, Arduino knows that Pin 13 is an output. If the Arduino gets unplugged or turned off somehow, the first thing that happens when the Arduino restarts (within the `setup()` function) is that Pin 13 is set as an output. In other words, you don't need to remind the Arduino what the individual pins do over and over. You'll put all of your `pinMode()`

functions inside `setup()` so that they run only once. Figure 4-33 shows the LED blinking.

> **Note**
>
> `setup()` happens once and only once.

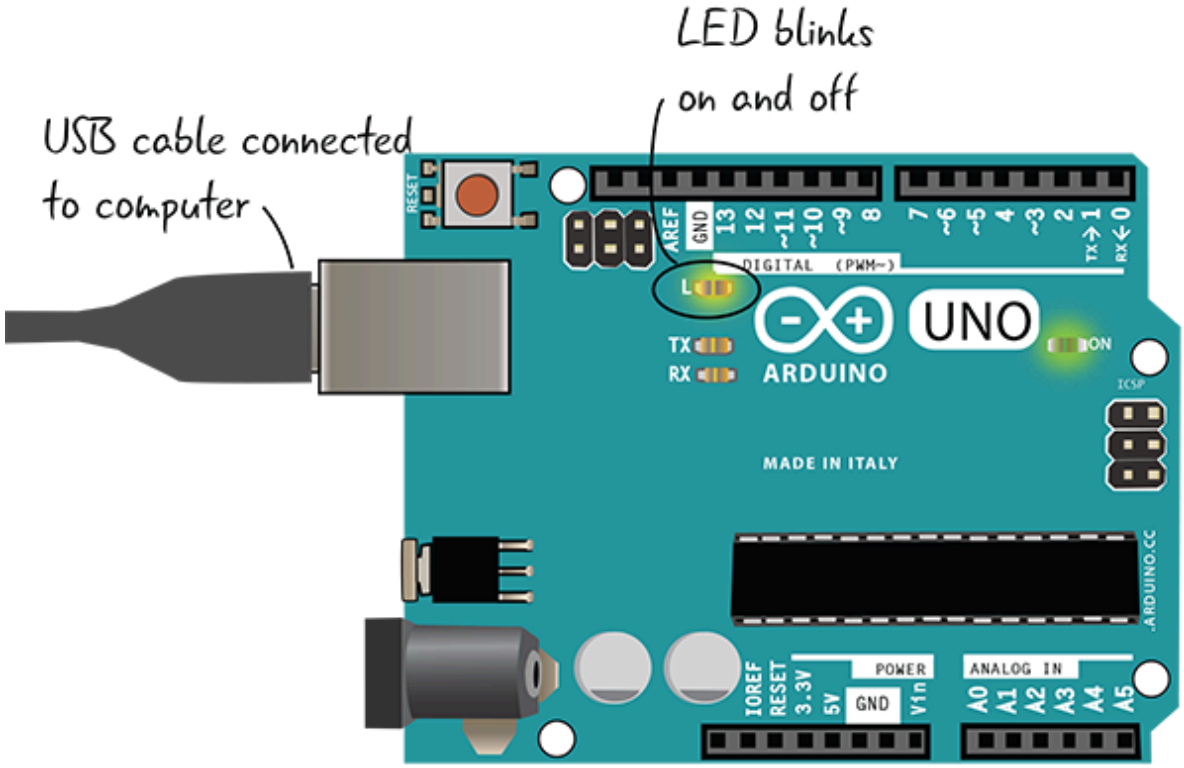


FIGURE 4-33: The LED blinking

QUESTIONS?

Q: Why should I bother putting comments in my code?

A: Sometimes when you are writing code, it's not obvious exactly what the code is doing, and it is very helpful to make notes to yourself for when you come back to a sketch later on. It is also helpful when you share your code or when you are working on a team.

Q: Does `setup()` always come first?

A: The Arduino knows to always run `setup()` first, once, and then continue on to running the `loop()` section of your code. In order to have a successful sketch, your code must include a `setup()` block.

Q: Can you explain again what it means that we set a pin using `pinMode()`?

A: By using `pinMode()`, we are instructing the Arduino that we plan to use a specific pin, in this case number 13 (also labeled as `LED_BUILTIN`), within our sketch. This is necessary for the Arduino to know which pins it will be controlling for each sketch.

Q: Do I always have to set pins as outputs?

A: No, you only set pins as outputs when you want to use them to turn things on and off. Pins can also be declared as inputs. We'll cover inputs in the next chapter.

Q: Is `pinMode()` always attached to `LED_BUILTIN`?

A: No, you have a lot of other pins on the Arduino you can use for your sketch. You are using `LED_BUILTIN` (Pin 13) right now

because it is the only pin that conveniently has an LED attached to it.

Q: Does it matter which pins I declare?

A: You should only declare pins you plan on using in the sketch. In the LEA4_Blink sketch, you only declare the `LED_BUILTIN`, Pin 13, because you know you're going to use that pin to turn on and off the LED.

Q: Setting the pin mode to output isn't the only thing I will do in `setup()`, right?

A: Right. There are a lot of other instructions to the Arduino that you want to run only once, and you will put them in `setup()`. We'll explain them later on.

LOOKING AT *LOOP()*: WHAT HAPPENS OVER AND OVER

Now that we have seen the `setup()` function from the LEA4_Blink sketch, let's take a look at the `loop()` function.

loop() function from the LEA4_Blink sketch

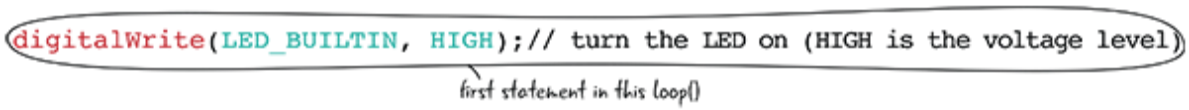
```
void loop() {
digitalWrite(LED_BUILTIN, HIGH);//turn the LED on (HIGH is the voltage level)
delay(1000);                    //wait for a second
digitalWrite(LED_BUILTIN, LOW);//turn the LED off by making the voltage LOW
delay(1000);                    //wait for a second
}
```

The `loop()` function contains the code that you want to have repeated over and over again. As long as the Arduino is running, this code will keep repeating, after the code in `setup()` has run once.

Note

`loop()` will continue running as long as the Arduino is on.

When you saw LEA4_Blink run on the Arduino, the LED light blinked off and on every second. The code in the loop creates this behavior. Let's take a close look at what's in `loop()` in our sketch line by line.



Note

In an Arduino sketch, write means to set a value.
`digitalWrite()` will set a value of `HIGH` or `LOW`.

The first statement of code instructions inside the `loop()` looks similar to the `pinMode(LED_BUILTIN, OUTPUT);` statement you saw in `setup()`. Again you'll be dealing with `LED_BUILTIN`, which is a label for Pin 13, since you declared in `setup()` that your sketch uses this pin. The `digitalWrite()` function in this context is used to set whether the pin is on or off. When you `write`, or set the value of the pin to `HIGH`, you are turning the pin completely on.

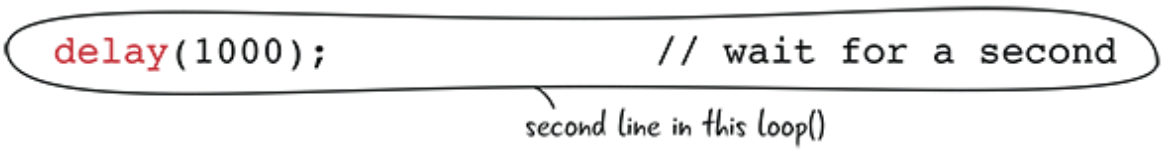
Note

`digitalWrite(pin #, HIGH)` will turn `pin #` on.

When the Arduino gets to this line in the `loop()`, it will turn on the LED attached to Pin 13. Let's next take a look at the second line.

LOOKING AT *LOOP()*: *DIGITALWRITE()* AND *DELAY()*

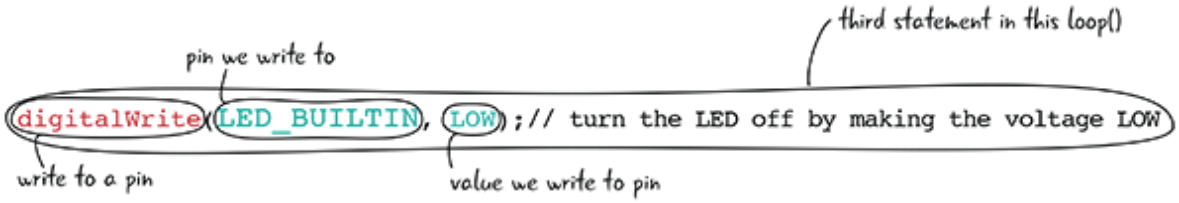
After you turn the pin on, you want to put a short `delay()` in the program. This `delay()` will pause your program, preventing the Arduino from reading the statement that follows for a short time. How long does `delay()` pause the program? That is up to you. `delay()` requires that you include between the parentheses the number of milliseconds (one thousand milliseconds equal one second) to wait. In this case, you have stated that the Arduino will wait one thousand milliseconds, or one second, before moving on to the next statement of the program.



Note

The `delay()` function stops the Arduino from doing anything for a short time.

Now let's look at the third line of the `loop()` function.



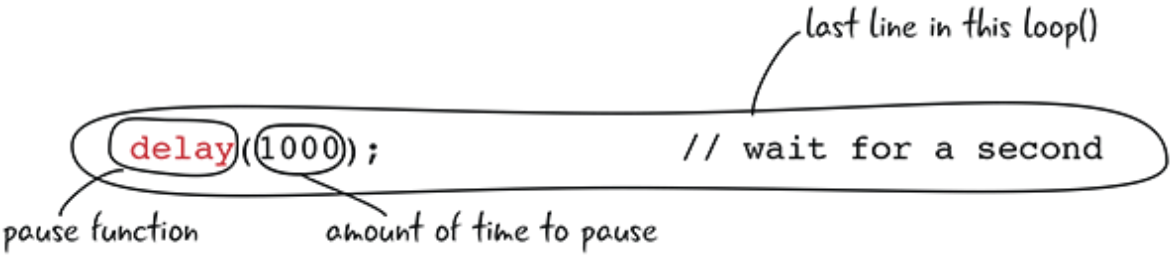
This third line of code instruction inside the `loop()` is nearly identical to the first line, `digitalWrite(LED_BUILTIN, HIGH);`, except that HIGH has been replaced with `LOW`. You are still focused on Pin

13, which is the only pin that you use in this sketch. As you learned with the first line, `digitalWrite()` determines whether the pin is on or off. Using the Arduino to write `LOW` sets the pin all the way down—in other words, off.

Note

`digitalWrite(pin #, LOW)` **will turn** `pin #` **off.**

Finally, let's look at the fourth and final line of the `loop()`. You'll put another pause in the program, this time for 1000 milliseconds, or one second. This makes it so that the LED stays off for a full second since the Arduino is paused for this time. The Arduino pauses for one second and then will go back to the first line of the `loop()` code again, repeating the cycle just described.



QUESTIONS?

Q: I can change the amount of time in a delay, right?

A: Absolutely. You'll see how to make the pauses longer and shorter, as well as make other modifications to the code in `loop()`, later on in this chapter.

LOOP(): LOOKING AT THE COMPLETE *LOOP()* FUNCTION

Here's all of the `loop()` code again, including the comments:

loop() function from the LEA4_Blink sketch

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

Again, as demonstrated in Figure 4-34, `loop()` is run continuously, and `setup()` is run once. Your `loop()` code will blink the light on and off until the Arduino is unplugged.

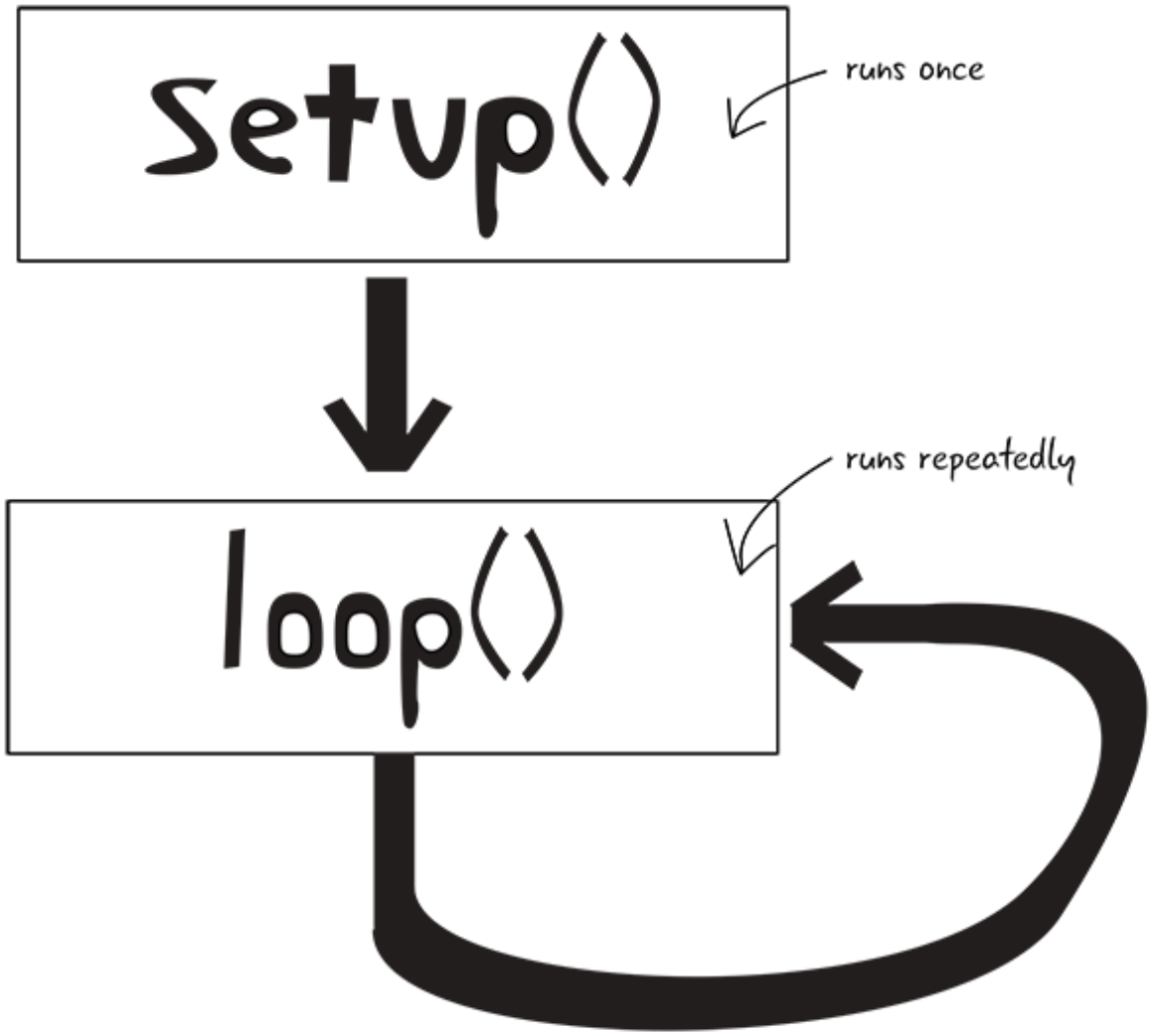


FIGURE 4-34: `setup()` and `loop()`

Note

Setting pin modes is always done in `setup()`. Anything you want to run more than once should be included in the `loop()` function.

Although the sketches you'll write throughout the book will become more complicated and include more lines of code, the basics laid out

in the LEA4_Blink sketch will continue to be your foundation for good code. Pins only need to be declared as inputs or outputs in `setup()`, and any code you want to happen more than once should be included in `loop()`. Remembering these two principles will help immensely as you get into more complex projects.

You've seen the basics of an Arduino sketch, and looked at how `setup()` and `loop()` functions work. After answering a few questions, we'll move on to reviewing schematics and look at the schematic for the Arduino. Then, we'll explain how to hook up your Arduino to a breadboard so you can run LEA4_Blink and light up an LED on the breadboard.

QUESTIONS?

Q: So whatever I put in `loop()` will keep repeating over and over again?

A: That's right. Just like the name, `loop()` keeps on looping through the same lines of code, over and over again.

Q: What does the `delay()` function do?

A: The `delay()` function specifies the amount of time that the Arduino is paused, or waiting idle. During this time everything stays the same, so if the light is on it stays on. With the `delay()` in this sketch, you can see clearly that the light turns on for one second, and then off again for one second.

Q: Will `digitalWrite()` always be in `loop()`?

A: No, it will only be there when you want to set a pin and whatever is attached to that pin, `HIGH` or `LOW`. In this case, you use `digitalWrite()` to turn the LED on or off.

Q: What exactly is a function again?

A: For now, think of a function as a way of organizing instructions to your Arduino. We'll explain more about them as you write more sketches.

Q: Semicolons, curly braces…it seems like there is a lot of punctuation in the sketch. How will I ever remember it?

A: It can be confusing when you start. Keep looking at the examples and see how the punctuation is used. Curly braces mark off a block of code, and semicolons mark the end of a line.

Q: Does the Arduino programming language have a reference guide online?

A: Yes. arduino.cc/en/Reference/HomePage is a great place to get more information about the language. You can use it to find out more about the code you use in this book, and to research your own projects after finishing the book.

A SCHEMATIC OF THE ARDUINO

Now that you've run your code and lit the LED on the Arduino board, you're going to attach your Arduino to a breadboard, build a circuit, and run your LEA4_Blink sketch again. You want to learn how to control external components with your Arduino, not just light up an LED on the Arduino itself, so you must attach a breadboard to hold the components.

To run LEA4_Blink on the Arduino attached to a breadboard, you won't need to make changes to your code. When you set `LED_BUILTIN` to `HIGH` in LEA4_Blink, it lights up the LED on the Arduino near Pin 13, and it will also set whatever is attached to Pin 13 (an LED) on a breadboard to `HIGH` (a.k.a. on).

Before you start to build your circuit, let's take a look at the schematic for it. Doing so helps you visualize the electronic relationships in the circuit.

Your schematics from now on will include a symbol for the Arduino. Figure 4-35 shows a schematic for the Arduino, with all the digital, analog, and power and ground pins labeled with their numbers or function, placed next to a drawing of the Arduino Uno for comparison. Don't worry about memorizing the pin numbers and functionality now—we'll explain more about the pins and their uses later.

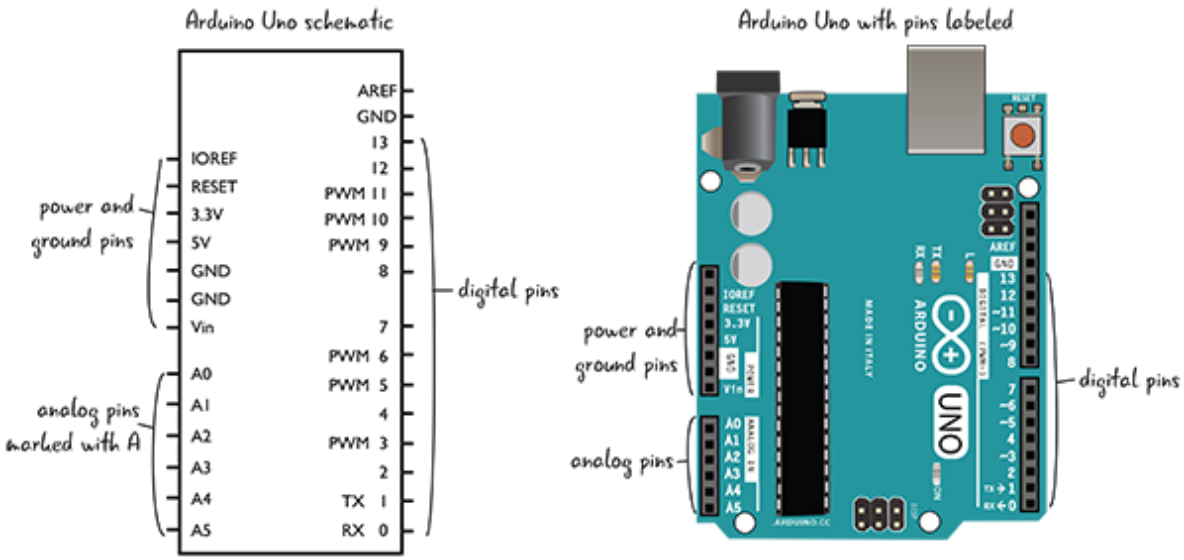


FIGURE 4-35: Arduino schematic and board

The schematic for the Arduino looks much more complicated than the other schematics you've seen previously. Its complexity reflects the number of connections possible with the Arduino hardware. Rather than try to cram this detailed Arduino schematic into the schematic of every circuit you build, you'll use a simplified version: a rectangle that represents the Arduino, with labels only for the components you're using in that circuit. Let's see a full schematic of the circuit you're going to build with the Arduino.

THE SCHEMATIC FOR YOUR CIRCUIT

For the sake of clarity, when you include the Arduino in your schematics, you're only going to label the pins that are attached to the circuit you're building. For example, Figure 4-36 shows the schematic for the circuit you're about to build. Only Pin 13, 5 volts, and ground are shown, as well as the LED and resistor.

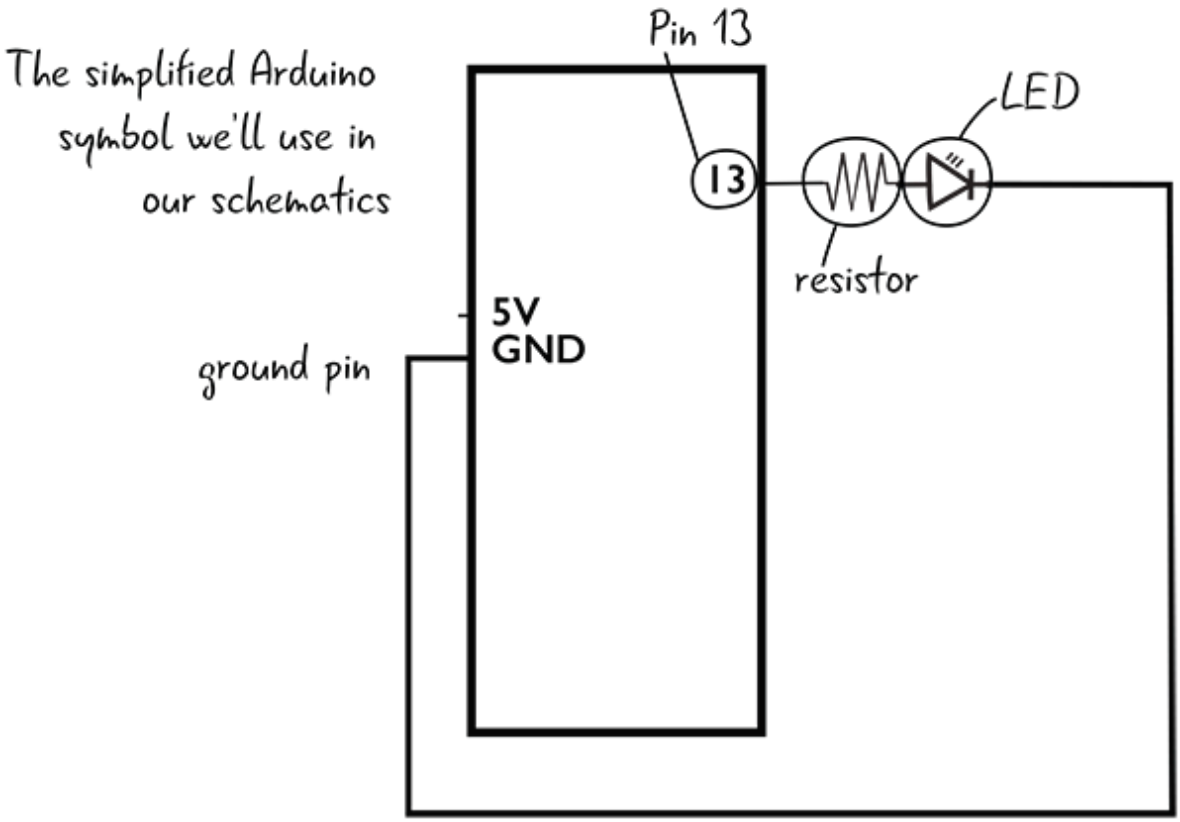


FIGURE 4-36: Schematic for LEA4_Blink circuit

Now that you've looked at the schematic, let's see how you're going to build the circuit. You'll start from the circuit you made in Chapter 3 (Figure 4-37).

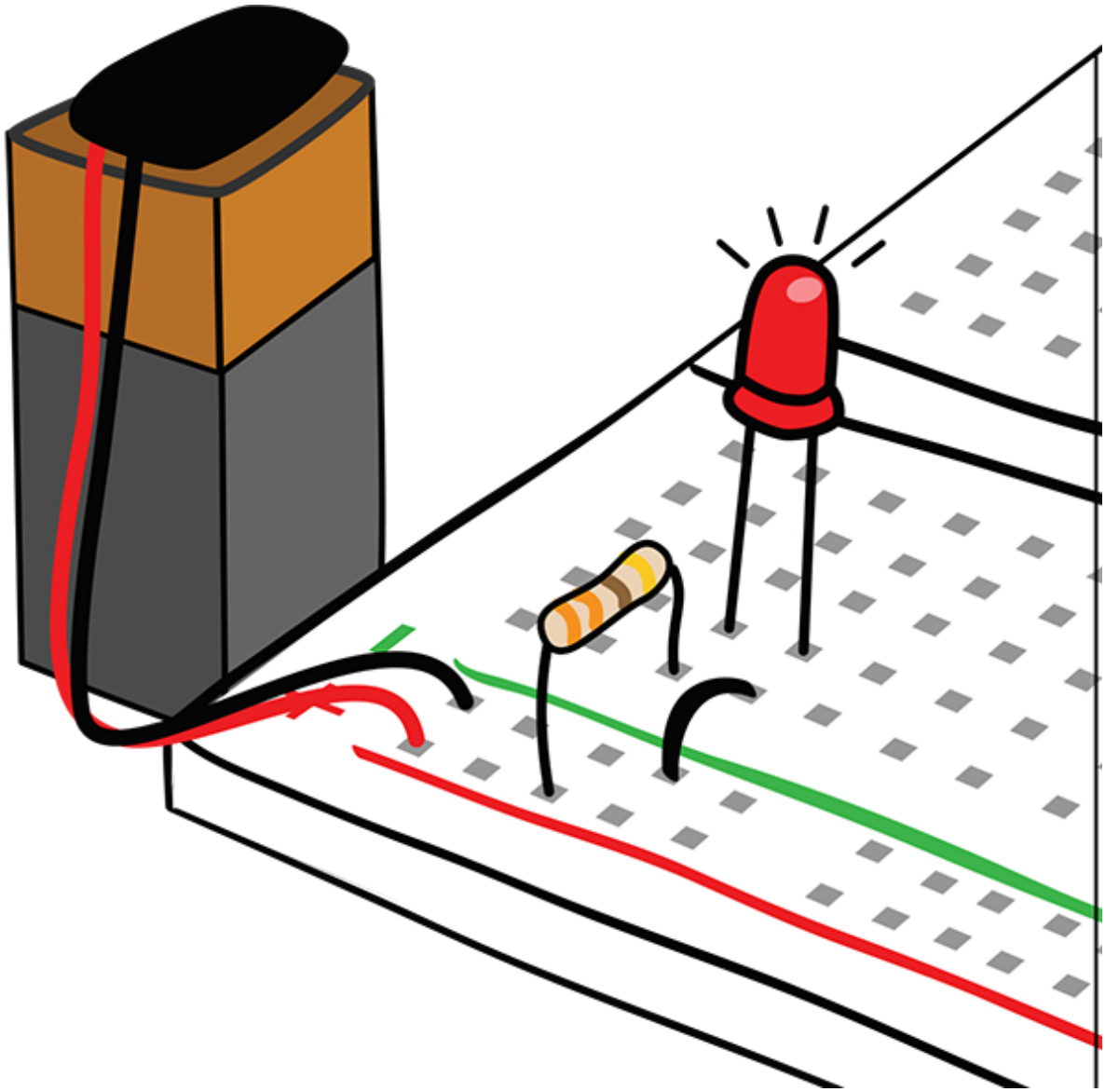


FIGURE 4-37: Circuit from Chapter 3

BUILDING THE BASIC CIRCUIT

Now that you've taken a look at the schematic, let's build the circuit. You're going to run your LEA4_Blink sketch and light up an LED on a breadboard. You're attaching a breadboard with a resistor and LED to the Arduino—the program that runs on the Arduino will not change. The Arduino will be the power source for your circuit when it is attached to a computer with a USB cable.

Warning

Remember, whenever you make adjustments to a circuit, your Arduino should *not* be attached to your computer.

You will need these parts:

LED (red)

220-ohm resistor (red, red, brown, gold). This is different from the one you used in the previous chapter.

Jumper wires

Breadboard

Arduino Uno

USB A-B cable

Computer with Arduino IDE

Figure 4-38 compares a drawing of this project to a schematic of the completed circuit. As you can see, the circuit uses a resistor and LED like the circuit you built in Chapter 3.

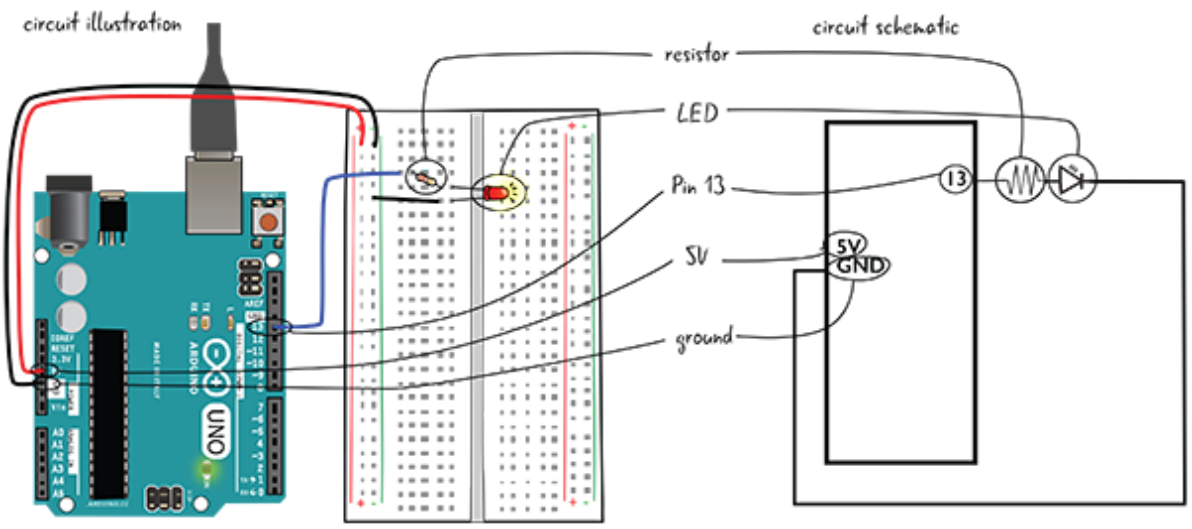


FIGURE 4-38: Labeled Arduino breadboard and schematic annotated

CONNECTING THE ARDUINO TO A BREADBOARD: FIRST STEPS

You want to build circuits with your Arduino, not just light up an LED on the Arduino board, so you're attaching it to a breadboard. How do you do that?

We first mentioned using the power and ground pins on the Arduino in Chapter 2. These two pins allow you to use electricity from the Arduino to power the components in your circuit, replacing the 9-volt battery you previously used.

To use the pins, start by attaching a jumper from the pin marked 5V to one of the power buses on your breadboard. Then attach a jumper from one of the pins marked GND (which stands for ground) to one of the ground buses on the breadboard. This is shown in Figure 4-39.

Warning

Make sure your computer is *not* attached to the Arduino when you are building a circuit.

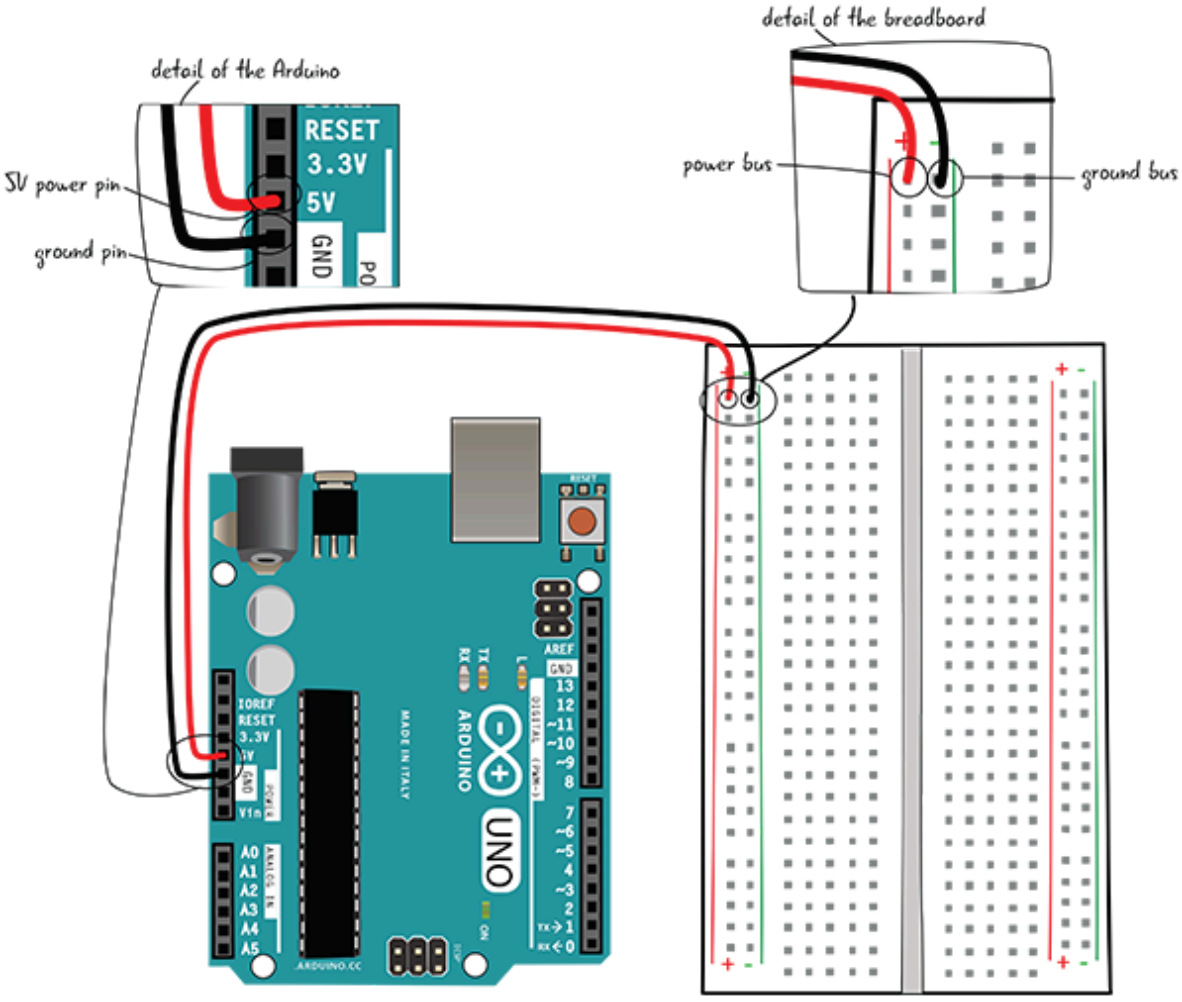


FIGURE 4-39: Attaching power and ground to the breadboard

It is standard procedure to attach both power (5V) and ground (GND) to the breadboard when attaching a breadboard to an Arduino. Even if you don't use the power right away, it can be handy to have later as you add more components to the circuit. In this

circuit, instead of using the 5V from the power pin, you'll be using Pin 13 to provide power for the LED.

BUILDING THE CIRCUIT STEP BY STEP: CONNECTING THE PIN AND RESISTOR

Now that the Arduino and breadboard are connected, connect Pin 13 on the Arduino board to a line of tie points in the breadboard with a jumper, as you see in Figure 4-40.

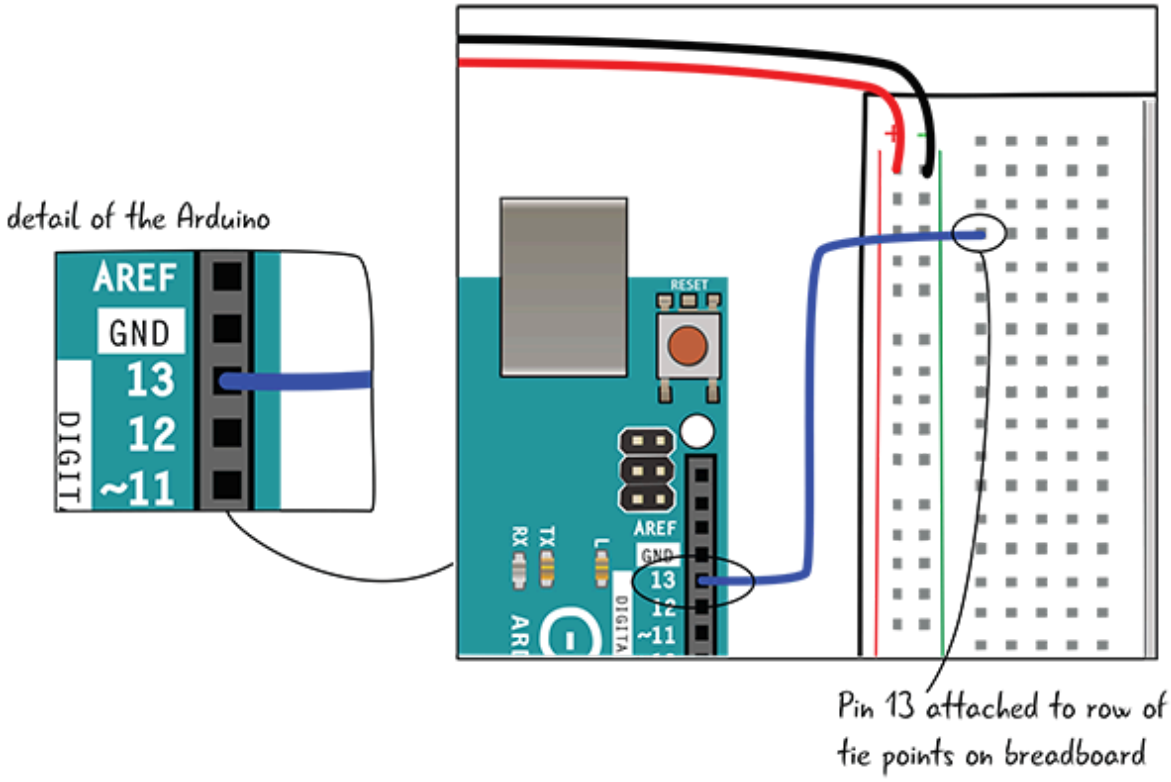


FIGURE 4-40: Adding a jumper from the pin to the breadboard

Next, put one end of a 220-ohm resistor (which has bands marked red, red, brown, gold) in the same row of tie points as the jumper from Pin 13. Put the other end into another row of tie points (Figure 4-41).

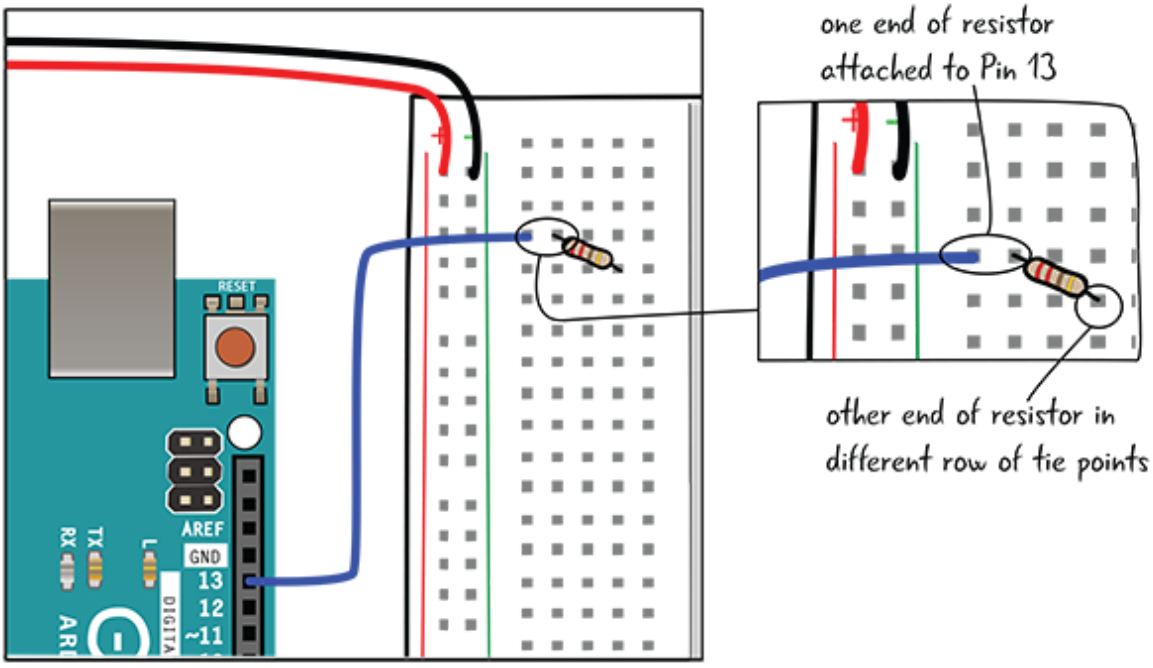


FIGURE 4-41: Adding the resistor to the circuit

BUILDING THE CIRCUIT STEP BY STEP: CONNECTING THE LED

Put the anode (long lead, positive end) of the LED in the same row of tie points as the other end of the resistor. Put the cathode (short lead, negative end) in another row (Figure 4-42).

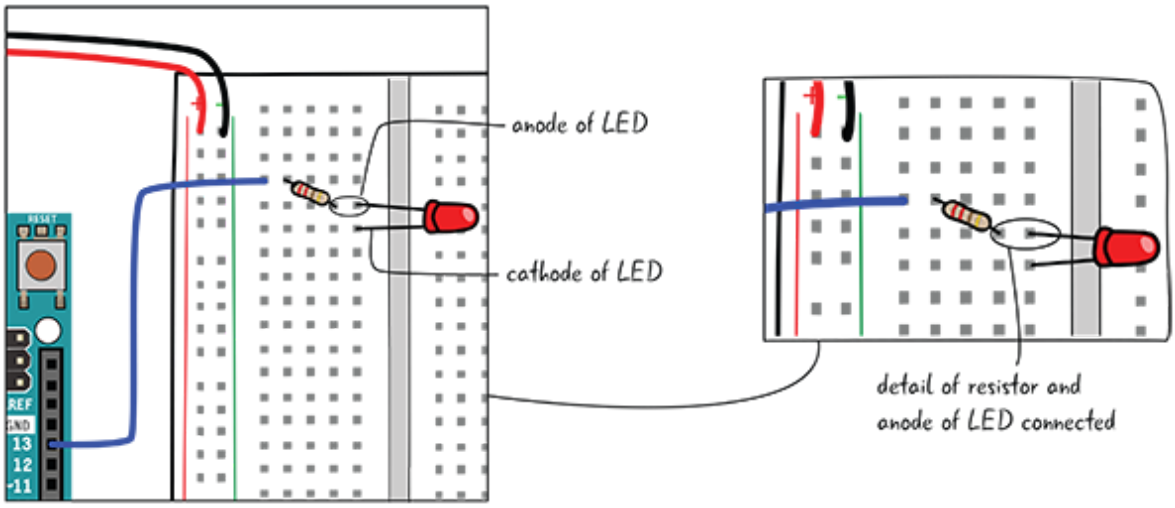


FIGURE 4-42: Adding the LED to the circuit

Next, add a jumper that connects the cathode (short lead, negative end) of the LED to the ground bus (Figure 4-43).

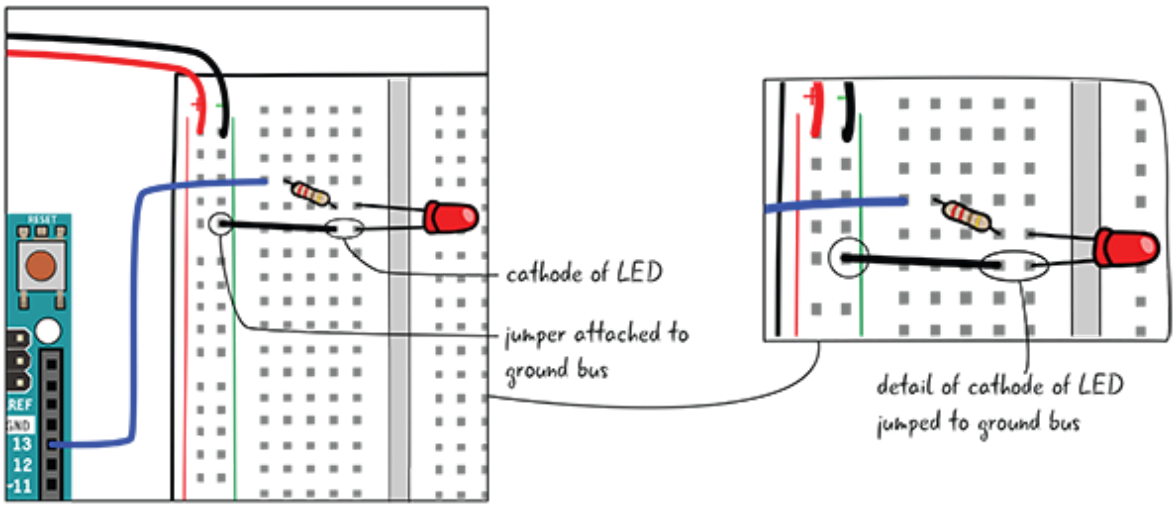


FIGURE 4-43: Adding a jumper from the LED to ground

BUILDING THE CIRCUIT STEP BY STEP: ATTACH TO YOUR COMPUTER

Finally, connect the USB cable that is attached to the computer to give your circuit power (Figure 4-44).

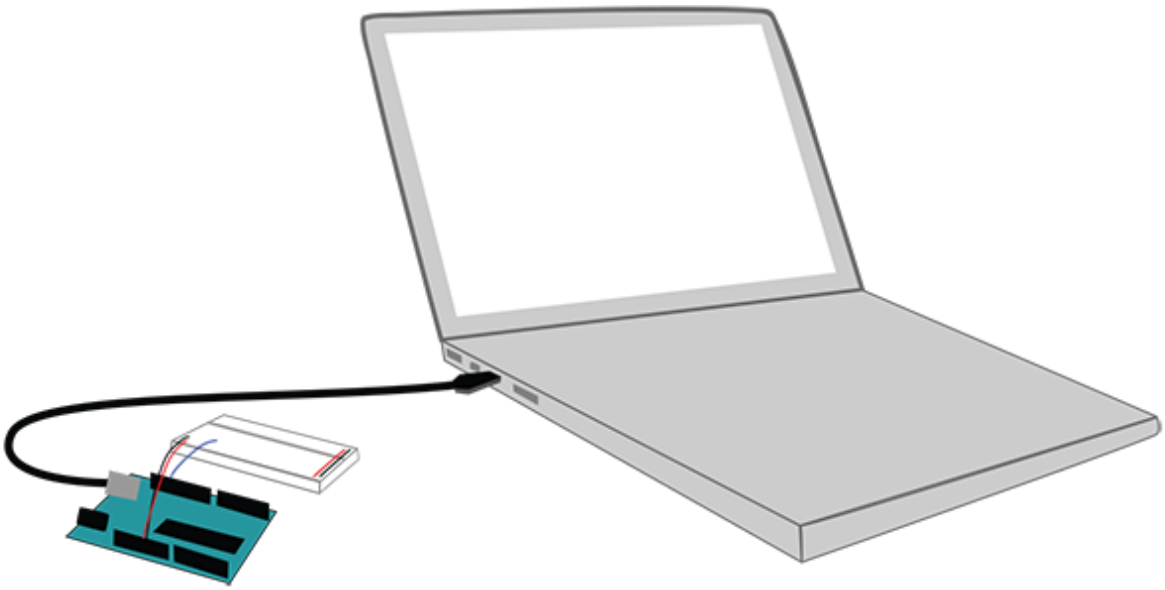


FIGURE 4-44: Attaching the Arduino to your computer

The LED should start blinking on the breadboard (Figure 4-45). Your circuit is like the basic circuit you created in the previous chapter, but now your LED flashes on and off, controlled by your Arduino, which is running the LEA4_Blink sketch. You have more control over your LED by using the Arduino; you have added the element of timing.

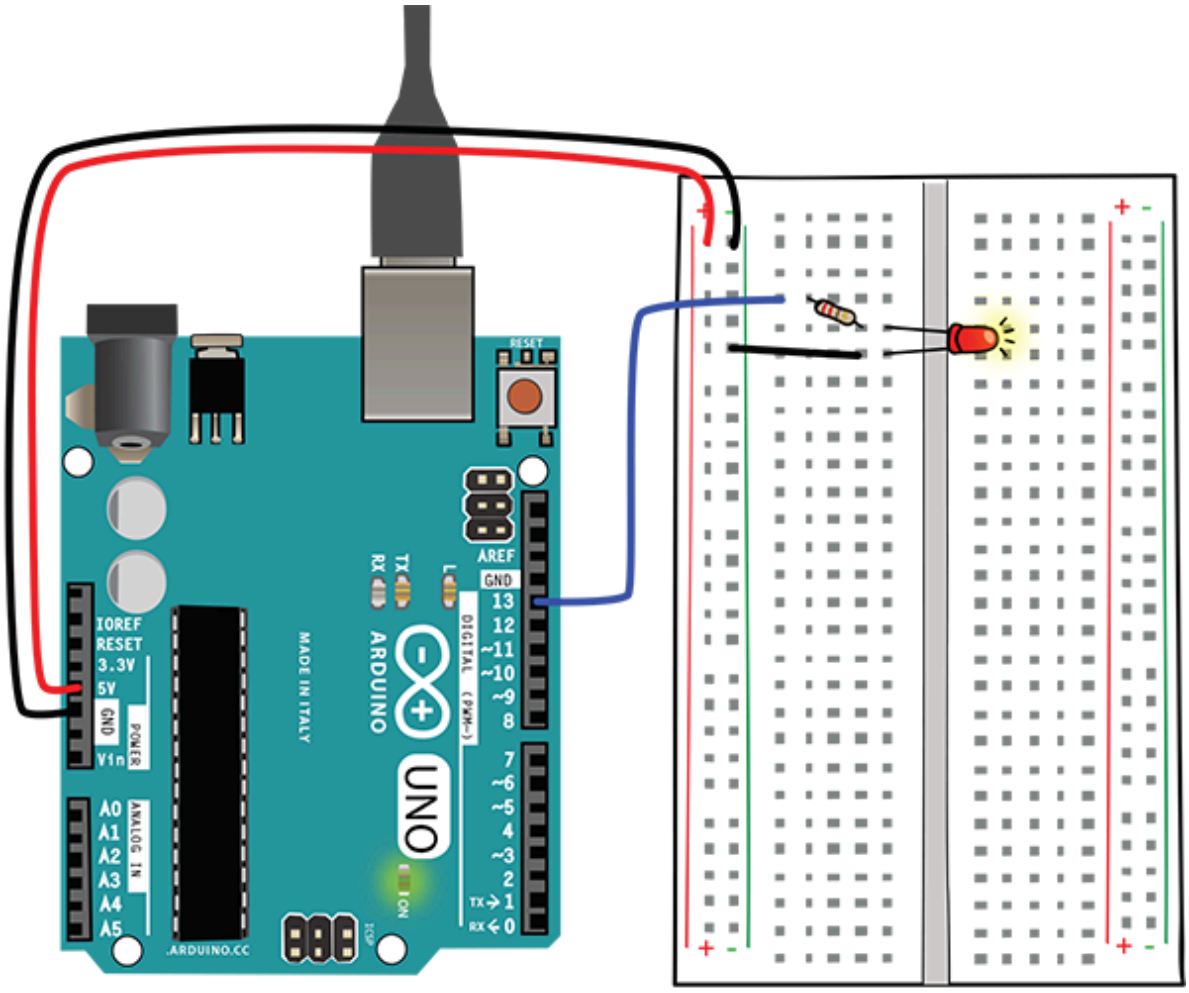


FIGURE 4-45: The blinking LED

QUESTIONS?

Q: My LED didn't light up—what's wrong?

A: Remember the section about debugging the circuit from Chapter 3? Check the continuity by looking carefully at the board or using your multimeter. Make sure the LED has the correct orientation. Also check that your jumpers are attached properly to the breadboard and to the Arduino.

Q: I didn't change the code in the LEA4_Blink sketch; why does this work?

A: The code in the LEA4_Blink sketch controls the `LED_BUILTIN` on the Arduino Uno. The tiny built-in LED is connected to the Arduino board on Pin 13, but the code in the sketch will also control any components that are attached to Pin 13.

Q: Tell me again why we connected 5 volts to the breadboard when it doesn't appear we're using it?

A: It is a convention to attach power and ground to the power and ground buses on a breadboard when you set it up. As you build more complex circuits, you'll eventually be using the power bus. This circuit gets the power from the pin on the Arduino.

SOS SIGNAL LIGHT: CREATING MORE COMPLEX TIMING

While the previous circuit ended up being very similar to the project in Chapter 3, you have accomplished something by hooking it up to the Arduino and discovering the possibilities of code. Earlier in this

chapter, you saw that you can make one light blink on and off with a few very simple lines, and the opportunity for complexity just grows from this basic starting point.

Now you'll work on the code, adjusting it to create an SOS signal light, a light that uses Morse code to convey an SOS message by a blinking light pattern. This is a pattern of three short light flashes, followed by three long flashes, and finally three more short flashes with a long pause at the end before the pattern repeats (Figure 4-46).

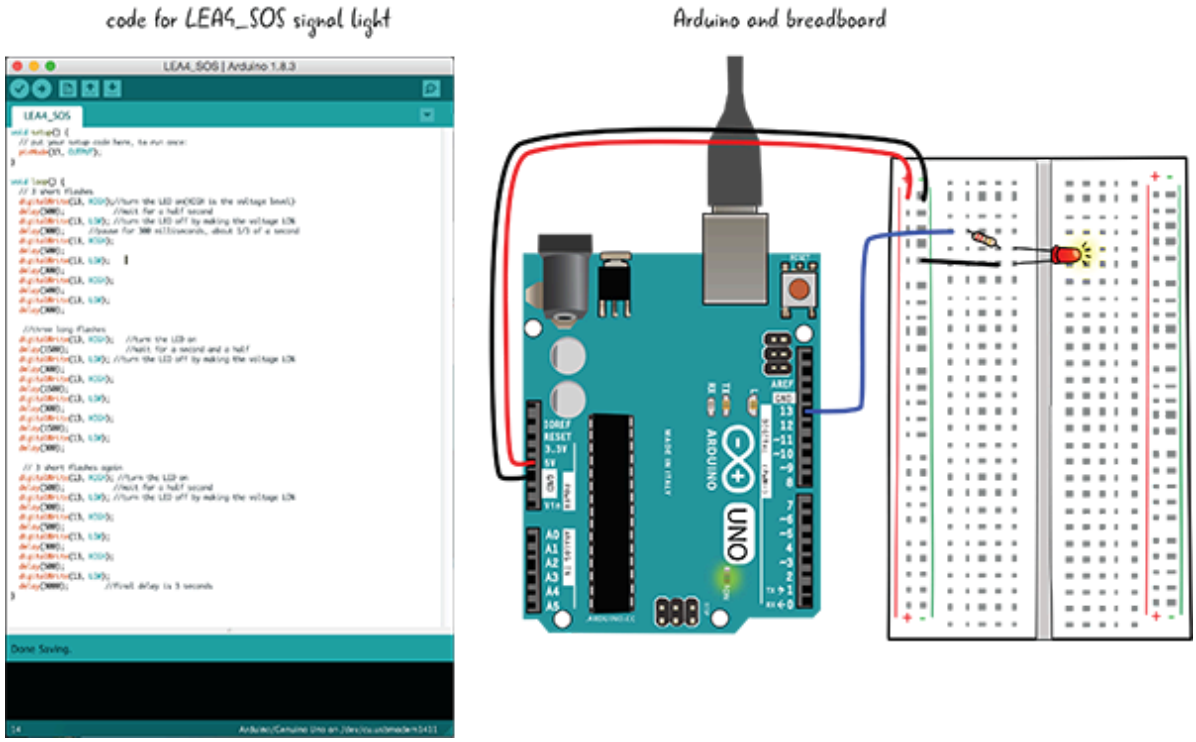


FIGURE 4-46: LEA4_SOS sketch and circuit

You can see by looking at Figure 4-46 that the hardware (the Arduino and breadboard with components) does not change at all. All the changes to make the LED blink in an SOS pattern will go in the sketch you write in the Arduino IDE. You don't need to disconnect your Arduino from your computer if you're adjusting the code—only when you're adjusting components on the circuit.

Note

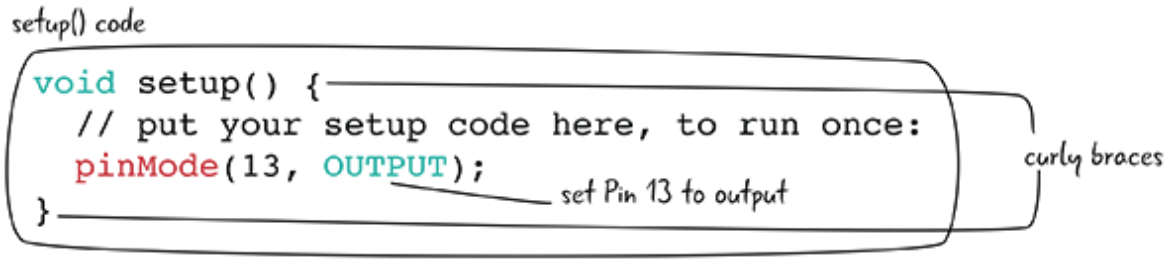
The Arduino can remain connected to your computer if you're adjusting code only, but not when you're changing the hardware.

SAVE SKETCH AND RENAME

Select Save As and rename your sketch LEA4_SOS. Some of this new sketch will have the same code you just used, and you'll be adding substantial new code. The code inside `setup()` will have one minor change and the code in `loop()` will become much longer. Let's review the LEA4_Blink code, then revise the code in `loop()`.

Reviewing and Revising Code: What Do You Change?

Let's first take a look at the `setup()` code. After a comment that tells you what the following line does, there is a line that sets `LED_BUILTIN`, connected to Pin 13 as an output. Instead of leaving it as `LED_BUILTIN`, you're going to change this `setup()` code to include the line `pinMode(13, OUTPUT)`.

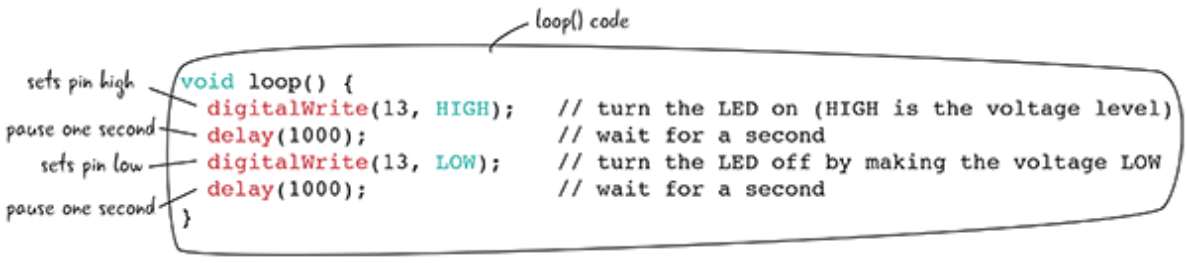


Note

You're changing `LED_BUILTIN` to 13 in the `pinMode()` in the `setup()` code.

Unlike the code in `setup()`, the code in `loop()` will be revised and added to extensively. Let's review the code from the LEA4_Blink sketch before you make changes.

The first line in the `loop()` sets `LED_BUILTIN` to `HIGH`, turning on the LED. `delay()` then pauses the Arduino—in this instance, for 1000 milliseconds, or one second. Next you set `LED_BUILTIN` to `LOW`, turning off the LED. `delay()` pauses again for 1000 milliseconds. Since the code in `loop()` repeats over and over again, the LED is blinking on and off.



Let's look at how you are going to revise `loop()`.

Adjusting *loop()* in the SOS Sketch

Your code for the SOS signal will be three short flashes of the LED followed by three long flashes, then another three short flashes, with a final pause before the code repeats again. You'll write the code for the three short flashes first. We'll look at all of it first, and then break it down line by line. We'll also reference Pin 13 by its number, replacing all mentions of `LED_BUILTIN` from the LEA4_Blink code.

THREE SHORT FLASHES ON AND OFF

After a comment that states what the code does, Pin 13 is set to `HIGH`, followed by a delay, then set to `LOW`, followed by a delay. This is repeated three times.

```
// 3 short flashes
digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
delay(500);               // wait for a half second
digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
delay(300);               //pause for 300 milliseconds, about 1/3 of a second
digitalWrite(13, HIGH);
delay(500);
digitalWrite(13, LOW);
delay(300);
digitalWrite(13, HIGH);
delay(500);
digitalWrite(13, LOW);
delay(300);
```

Let's take a closer look. The first line of code inside `loop()` will stay the same as in your LEA4_Blink sketch. As you have seen, this line sets Pin 13 to `HIGH`.

set Pin 13 to high

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

You're going to make an adjustment to the next line of code. Remember, the `delay()` function creates a pause, measured in milliseconds. In your original sketch you paused for 1000 milliseconds, or one second. You want a shorter pause now, 500 milliseconds, or half a second. Let's change the comments to reflect what your code is doing.

pause for 1/2 second

```
delay(500);               // wait for a half second
```

Your next line will set the pin `LOW`, or turn off the LED. You can leave this line as it is, since there is no need to change it from the LEA4_Blink sketch.

set Pin 13 to low

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

However, you'll make a change in the number of milliseconds in `delay()`. In your LEA4_Blink sketch, the delay was 1000 milliseconds, or one second. Now you'll pause 300 milliseconds, about a third of a second. You'll adjust the comments as well.

pause for 300 milliseconds

```
delay(300);              //pause for 300 milliseconds, about 1/3 of a second
```

Here is the complete cycle:

```
digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
delay(500);               // wait for a half second
digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
delay(300);               //pause for 300 milliseconds, about 1/3 of a second
```

You want to repeat turning on and off the LED three times. Let's first add a comment indicating what this part of the code does, then copy two more cycles of turning on and off. Here is the code again:

```
// 3 short flashes
digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
delay(500);               // wait for a half second
digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
delay(300);               //pause for 300 milliseconds, about 1/3 of a second
digitalWrite(13, HIGH);
delay(500);
digitalWrite(13, LOW);
delay(300);
digitalWrite(13, HIGH);
delay(500);
digitalWrite(13, LOW);
delay(300);
```

Now let's look at the code for the longer flashes.

ADDING THE THREE LONG FLASHES ON AND OFF

The three long flashes section is very similar to the short flashes section. After the pin is set to `HIGH`, the `delay()` function pauses for

1500 milliseconds, or one and a half seconds, keeping the LED turned on. Let's look at all of the long-flash code first. A comment states what the code immediately following does.

```
// 3 long flashes
digitalWrite(13, HIGH);   // turn the LED on
delay(1500);              // wait for a second and a half
digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
delay(300);
digitalWrite(13, HIGH);
delay(1500);
digitalWrite(13, LOW);
delay(300);
digitalWrite(13, HIGH);
delay(1500);
digitalWrite(13, LOW);
delay(300);
```

Again you have a repeating cycle of setting the pin to `HIGH`, pausing, setting the pin to `LOW`, pausing, three times. First you set the pin to `HIGH`.

set Pin 13 to high

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

Then you pause with the `delay()` function, this time for 1500 milliseconds, or a second and a half. The comments have also been changed to reflect the adjusted amount of time.

pause for 1 1/2 seconds

```
delay(1500);              //wait for a second and a half
```

Just as in the code for the short flashes, you must set the pin to `LOW`, then pause with `delay()`.

set Pin 13 to low

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

You'll then use the same number of milliseconds as `delay()` between the short flashes, 300 milliseconds.

pause for 300 milliseconds

```
delay(300);              //pause for 300 milliseconds, about 1/3 of a second
```

Again, you're creating a cycle that is going to repeat. After the last short flash cycles, you'll make the pause last longer to make each SOS signal discrete. Let's look at that, and then look at all the code in `loop()` together.

This final line of code in `loop()` pauses the Arduino for 3000 milliseconds, or 3 seconds. This follows a line that has set Pin 13 to `LOW`. You want a longer pause between each SOS signal to make sure viewers can distinguish between cycles.

final lines in loop()

```
digitalWrite(13, LOW);
delay(3000);              //final delay is 3 seconds
```

QUESTIONS?

Q: I didn't change my code from `LED_BUILTIN` to 13. Why does it still work?

A: `LED_BUILTIN` is the same as Pin 13, so even if you switch between the two, your sketch will still work. It is best to pick only one so as not to confuse what is happening in your sketch.

ALL OF THE SOS *LOOP()* CODE

Now let's look at all of the code in `loop()`. It is long, so we're breaking it up into sections.

```
void loop() {
// 3 short flashes
 digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
 delay(500);               // wait for a half second
 digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
 delay(300);               //pause for 300 milliseconds, about 1/3 of a second
 digitalWrite(13, HIGH);
 delay(500);
 digitalWrite(13, LOW);
 delay(300);
 digitalWrite(13, HIGH);
 delay(500);
 digitalWrite(13, LOW);
 delay(300);
```

loop() declaration and start curly brace

3 short flash code

```
// 3 long flashes
digitalWrite(13, HIGH);   // turn the LED on
delay(1500);              // wait for a second and a half
digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
delay(300);
digitalWrite(13, HIGH);
delay(1500);
digitalWrite(13, LOW);
delay(300);
digitalWrite(13, HIGH);
delay(1500);
digitalWrite(13, LOW);
delay(300);
```

3 long flash code

```
// 3 short flashes again
 digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
 delay(500);               // wait for a half second
 digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
 delay(300);               //pause for 300 milliseconds, about 1/3 of a second
 digitalWrite(13, HIGH);
 delay(500);
 digitalWrite(13, LOW);
 delay(300);
 digitalWrite(13, HIGH);
 delay(500);
 digitalWrite(13, LOW);
 delay(3000);          //final delay is 3 seconds
}
```

3 short flash code

final delay is 3000 milliseconds

curly brace closes the loop code

After you've written your code for the SOS signal light and saved it, click the Verify button to check for errors (Figure 4-47).

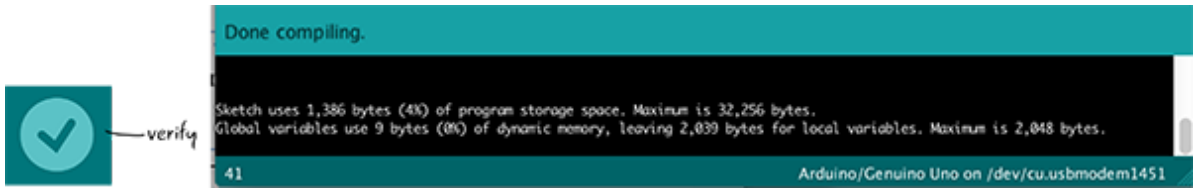


FIGURE 4-47: Successful verify

If it is okay, make sure your computer is attached to your Arduino, and that you have the correct board and port selected. Then click the Upload button to upload your code to the Arduino (Figure 4-48).

FIGURE 4-48: Successful upload

What does the LED look like now on the board?

SOS SIGNAL LIGHT FLASHES ON AND OFF!

Your LED should now be flashing an SOS signal: three short bursts, followed by three long flashes, three short bursts again, a 3-second pause, then the whole pattern starting over and over again. There are other, more efficient ways to write the code, but for now we want you to make adjustments and understand what the code is doing by seeing the results in your circuit (Figure 4-49).

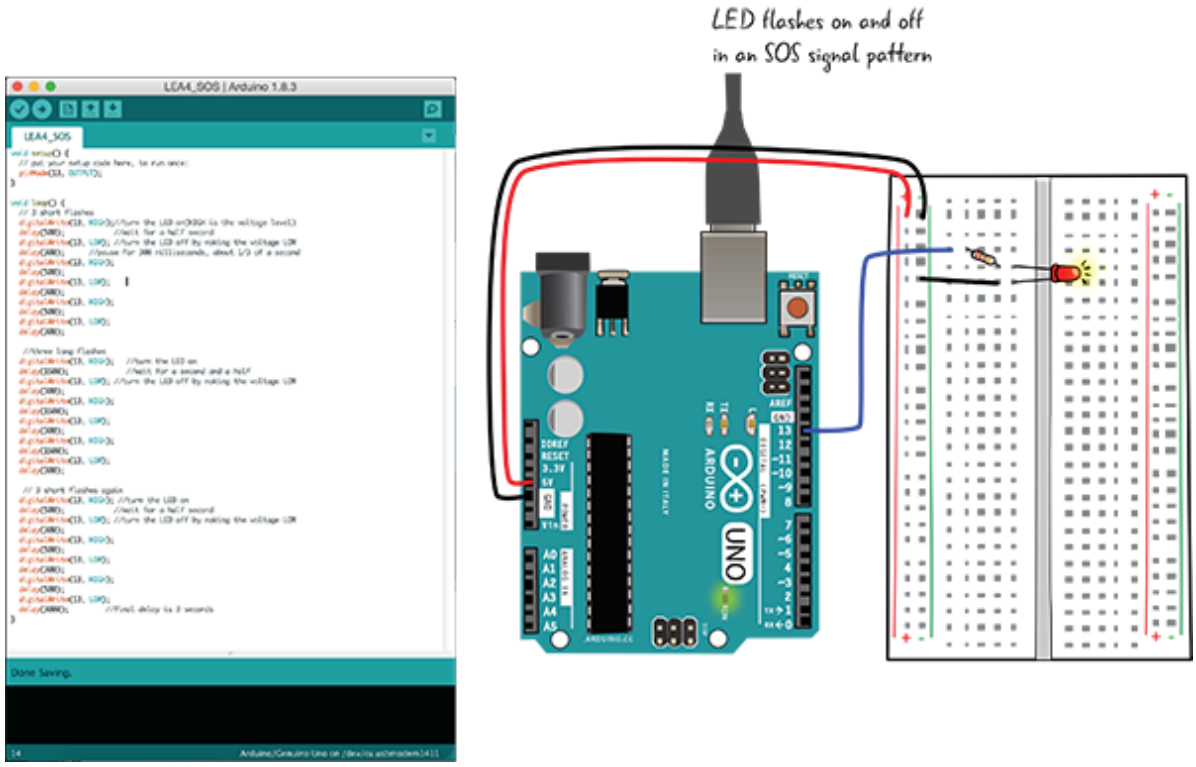


FIGURE 4-49: LED flashing SOS

SUMMARY

You've set up the Arduino IDE, learned how to verify and upload code, seen how to attach a breadboard to an Arduino, and explored writing a sketch in the Arduino programming language. You can download the code for LEA4_SOS from https://github.com/arduinotogo/LEA/blob/master/LEA4_SOS.ino.

In the next chapter, you'll learn more about writing code in the Arduino programming language and how to attach different types of components to a circuit.

5

ELECTRICITY AND METERING

What are voltage, current, and resistance? How are they related? And why should you care?

In this chapter, you'll learn about voltage, current, and resistance and how they interact with each other. This will help you understand how your circuits are working and how to make adjustments to them. You'll also learn how to use the multimeter to measure these properties. This knowledge will help you debug your circuits and also get you on your way to designing and building your own projects from scratch.

UNDERSTANDING ELECTRICITY

Electricity is the flow of electrons through a material, as you can see in Figure 5-1. In the projects in this book, electrons flow through carefully arranged and specified paths—through our circuits.

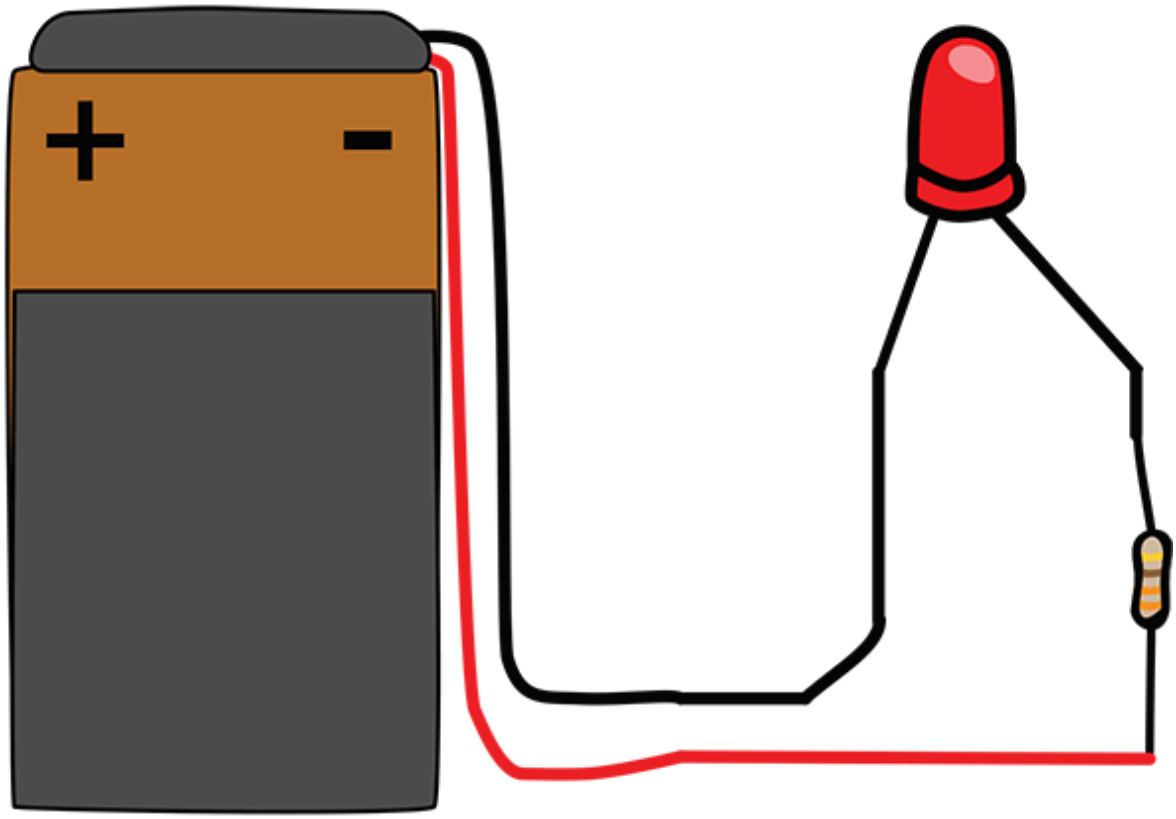


FIGURE 5-1: Electricity flows through a circuit.

Electricity has three main properties: voltage, current, and resistance. In this chapter, you'll see how these properties interact with each other, in a fundamental relationship called *Ohm's law*. You'll also learn how placing components in different arrangements affects the electrical properties in a circuit.

Why are we looking at electrical properties, and not just building more circuits with our Arduino and other components? If you don't understand a bit about how these properties work in a circuit, it will be very difficult to move on to building your own circuits after you've completed all the projects in this book. Also, without some understanding of these properties, troubleshooting your projects is next to impossible. In this chapter, you'll learn more techniques for debugging your projects.

MEASURING ELECTRICAL PROPERTIES WITH YOUR MULTIMETER

Remember the multimeter (Figure 5-2) from Chapter 3, "Meet the Circuit"? You learned how to set it up to test for continuity (whether your components are connected to each other). The multimeter helps you debug problems in a circuit. By testing for continuity, for example, you can verify that your circuit is a complete loop. In this chapter, you'll learn how to use the multimeter to measure voltage, current, and resistance. Why do you need to do this? Testing voltage will help you analyze problems with your circuits; for example, is your circuit getting voltage? How much voltage is each of your components consuming?

> **Note**
>
> Understanding how voltage, current, and resistance interact in a circuit helps you troubleshoot your projects as well as build new circuits.

FIGURE 5-2: The multimeter

If you're going to use a multimeter to test electrical properties in a circuit, you'll first need to build a circuit. Let's start with a basic circuit that contains one LED, a resistor, a breadboard, and an Arduino. You won't write an Arduino sketch this time; instead, you'll simply use the Arduino as a power source. You'll check the voltage coming out of

the Arduino, and then test the voltage across each component. You'll then add a second LED to the breadboard and see how the electrical properties of the components change depending on if you place them in a series or in a parallel arrangement. We'll explain exactly what we mean by all of this shortly.

BUILD THE CIRCUIT STEP BY STEP

To build the basic circuit described in this chapter, you'll need the following parts:

1 red LED

1 220-ohm resistor (red, red, brown, gold)

Jumper wires

Breadboard

Arduino Uno

USB A-B cable

Computer

This circuit is quite similar to the circuit you built in Chapter 4, "Programming the Arduino." The one difference is that you aren't powering the LED from a pin on the Arduino but from the 5-volt power bus on the breadboard.

BUILD!

As we said earlier, there is one major difference in this circuit, shown in Figure 5-3, from the one you built in Chapter 4: you aren't connecting it to a digital pin on the Arduino. Instead, you're going to get power from the power bus on the breadboard. Remember, the power bus is connected by a red jumper to the pin marked 5V (for 5 volts) on the Arduino.

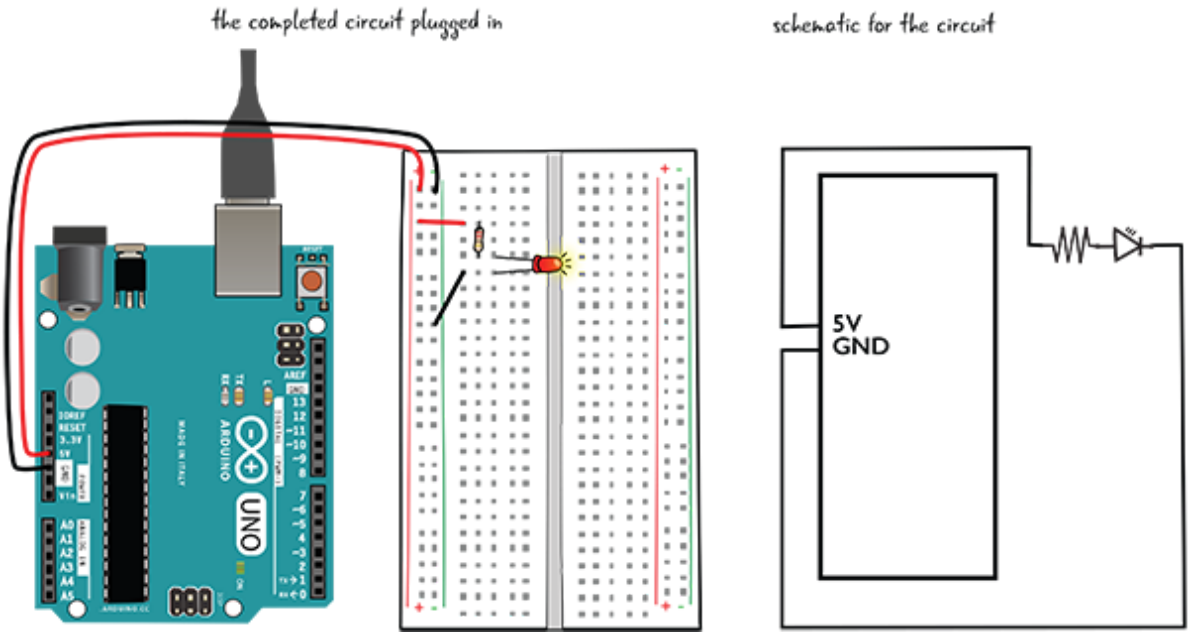


FIGURE 5-3: The circuit with the schematic

Before we start building, let's take a look at the block of power and ground pins, shown in Figure 5-4. There is a pin marked 5V, and also one marked 3.3V, as well as pins marked GND for ground.

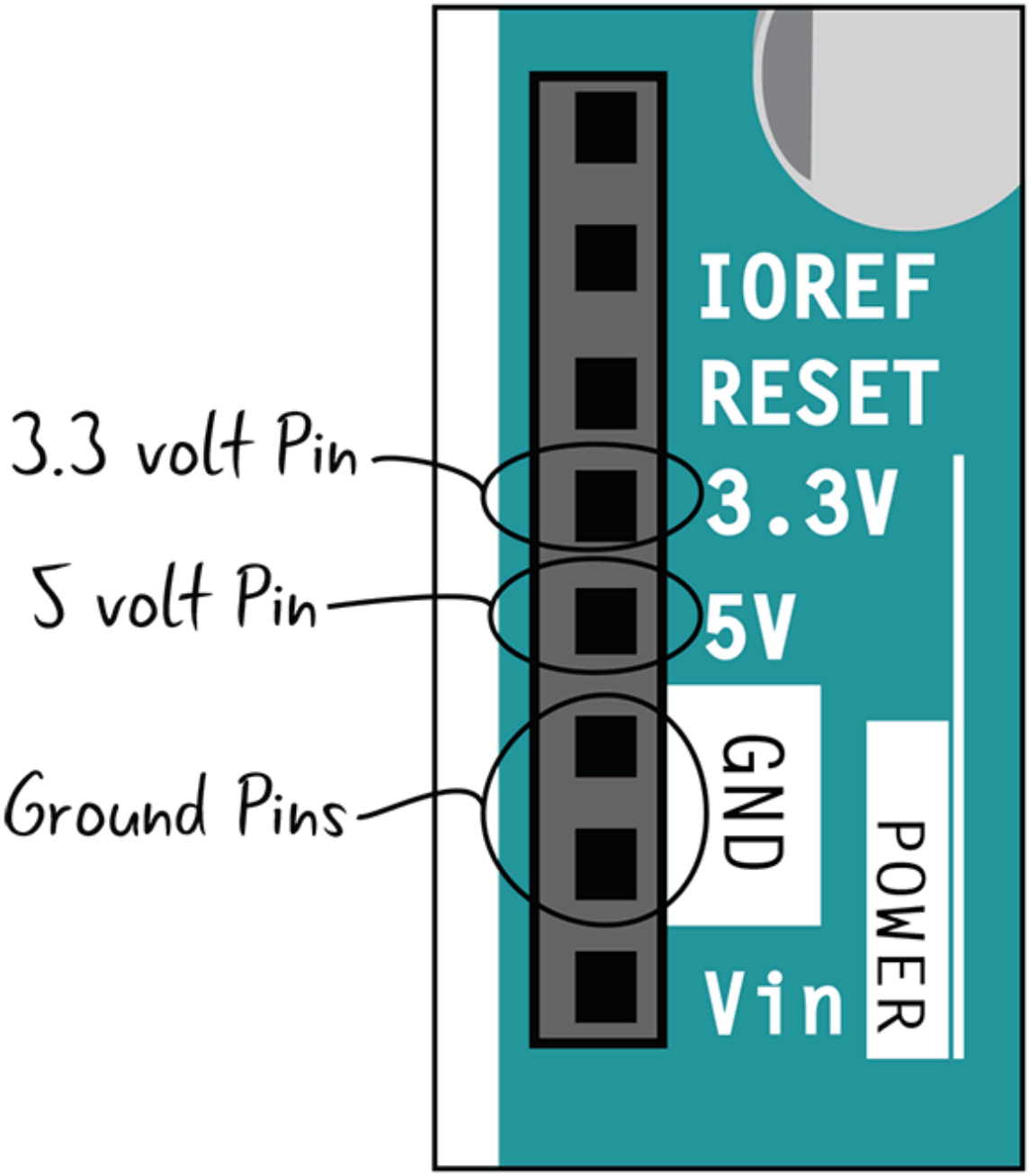


FIGURE 5-4: Power and ground pins on Arduino

Although we have been building our circuits using 5V, you can also use 3.3V to build some circuits with components that require less voltage. The 3.3V port works the same as the 5V except it puts out a lower amount of voltage.

Here are the steps to build the circuit, shown in Figure 5-5:

Attach one end of a jumper to the 5V pin on the Arduino, and the other end of the jumper to the power bus on your breadboard (that's the column marked with a red +).

Grab another jumper and attach one end to the GND pin on the Arduino, and the other end to the ground bus on your breadboard (that's the column marked with the green –).

Connect a jumper from the power bus to a row of tie points.

Connect one lead of a 220-ohm resistor to the same row of tie points. The other end goes in another row of tie points.

Connect the anode (long leg) of the LED to the other lead of the resistor.

Attach a jumper from the cathode (short leg) of the LED to the ground bus.

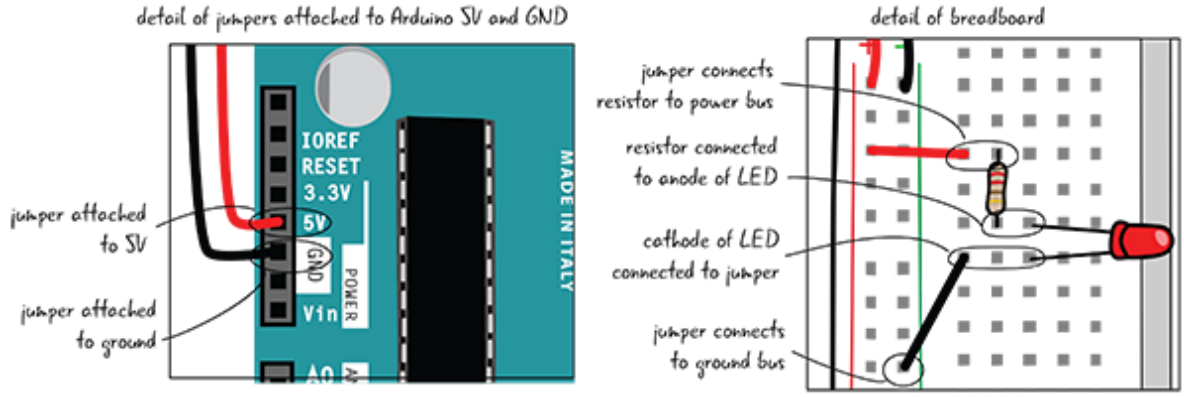


FIGURE 5-5: A circuit with details of power and ground pins on Arduino and the components on the breadboard

When you have the circuit built, use the USB cable to attach your Arduino to your computer. We aren't going to write a sketch; you're just using your computer as a power source for the Arduino.

DEBUGGING THE CIRCUIT

If your LED lights up when you attach the USB cable to your computer, you can skip to the next page. If not, let's troubleshoot, or "debug" the circuit.

You learned about debugging in the earlier chapters. As a reminder, it is defined as the process by which you methodically check your project to eliminate any issues that might be causing problems.

Check that power and ground are connected to the breadboard buses and the correct ports on the Arduino. Figure 5-6 shows them connected improperly.

Check that the LED is oriented correctly (anode connected to resistor that is connected to power, cathode attached to jumper that is attached to ground). Figure 5-7 shows what it should look like.

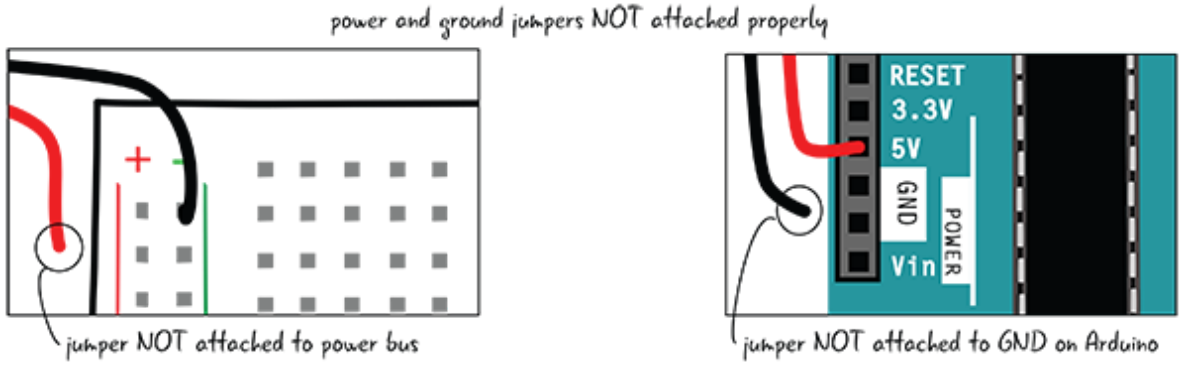


FIGURE 5-6: Power and ground on Arduino and breadboard improperly attached

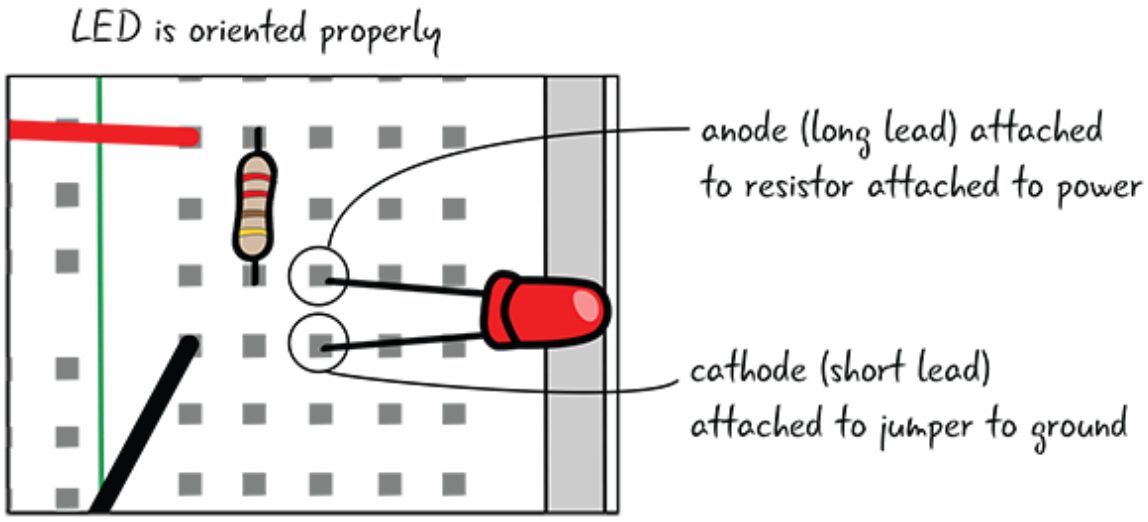


FIGURE 5-7: Check the orientation of the LED.

Check the continuity; the leads of the components that are supposed to be connected need to be in the same row of tie points. Figure 5-8 shows a circuit where the components are not connected.

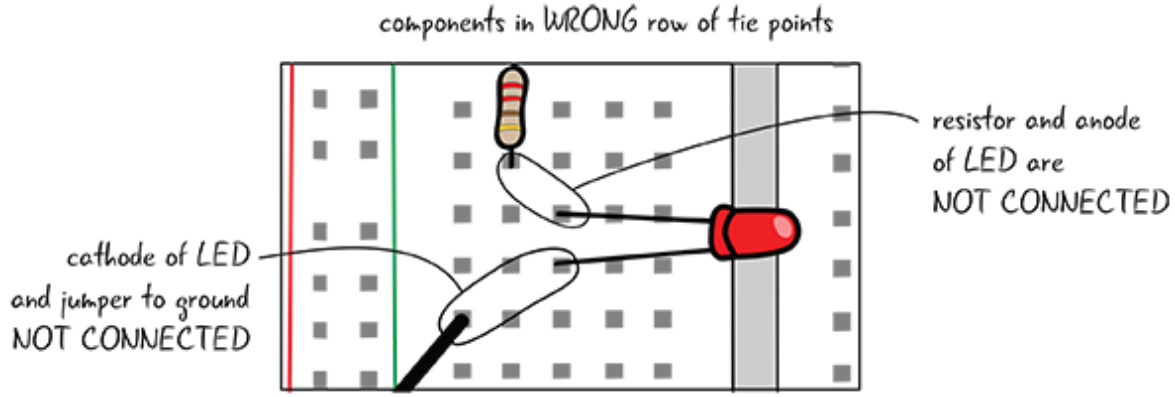


FIGURE 5-8: Components *not* connected

Now that you have your circuit fully functional, let's discuss how electricity is flowing through the circuit.

QUESTIONS?

Q: Why are you putting so much emphasis on debugging?

A: In any electronic project, many things can potentially go wrong. It is smart to maintain a comprehensive approach to error checking.

ELECTRICITY: AN OVERVIEW

Electricity is the flow of electrons through a material. Electricity requires a closed loop to flow from beginning to end. Your circuits create a closed loop with conductive lines and components. The electricity follows the paths of the circuit. Figure 5-9 shows the path of electricity in the circuit you just built.

Note

This discussion of electricity is simplified. We wanted to reduce the complexity of explanations to better fit with the small-scale electronics projects we're building. This is not an adequate primer for a complete understanding of electricity or of complex electronic theory.

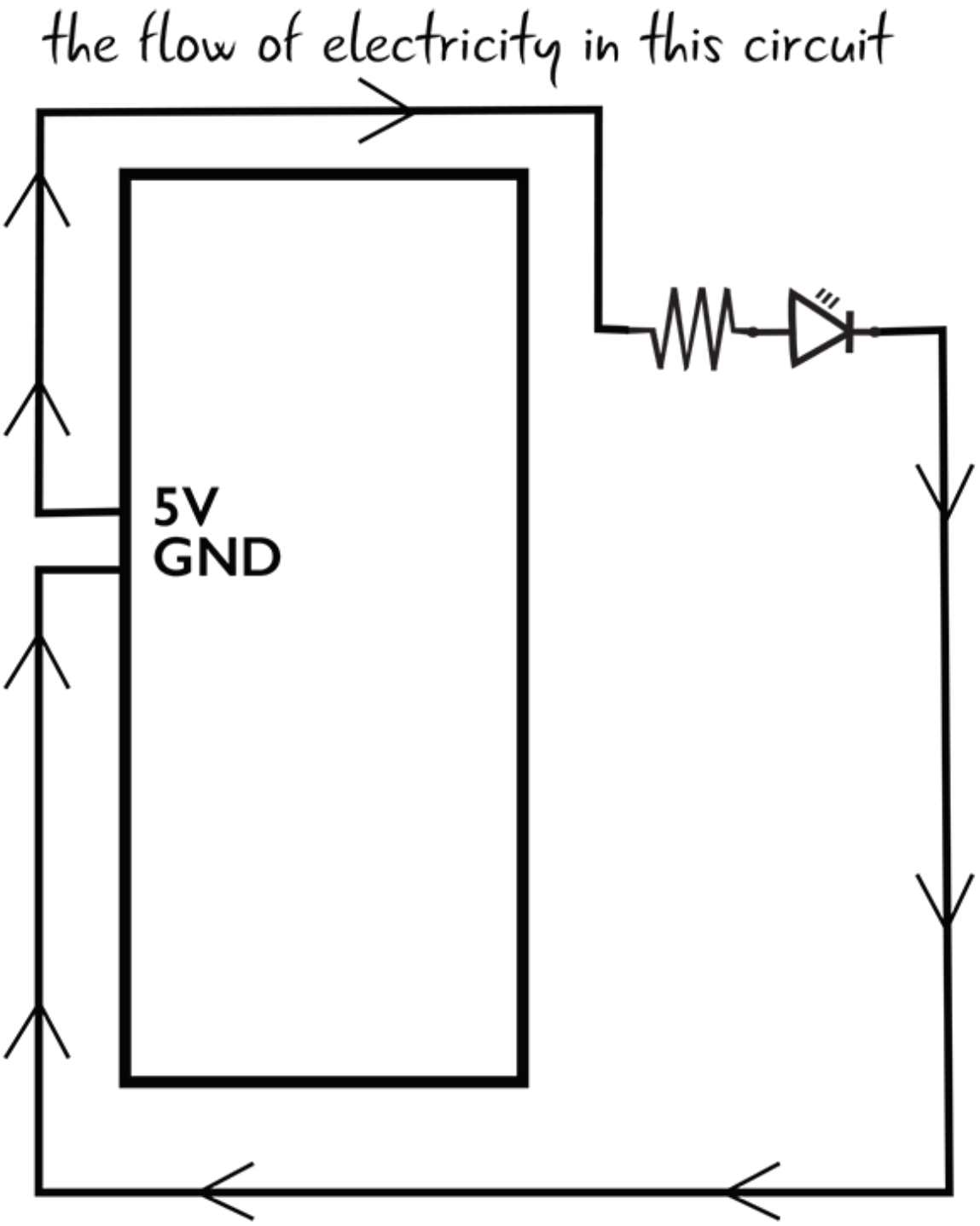


FIGURE 5-9: Schematic with electrical flow indicated

HOW DOES ELECTRICITY BEHAVE?

Materials can be broken down into two different types. The first type of material is *conductors*, which are good at letting electricity flow. Wires are made of metal because it is a good conductive material. The second type of material is *insulators*, which resist the flow of electricity. Rubber is one example of an insulator.

> **Note**
>
> Conductors let electricity flow, whereas insulators restrict the flow.

AC AND DC CURRENT

There are two different types of electrical flow: alternating current (AC) and direct current (DC). The electricity that comes out of your wall socket is AC, whereas our Arduino and many small electronic projects and components use DC. In alternating current, represented by the drawing in Figure 5-10, the flow of electricity changes direction. In direct current there is only one direction to the flow.

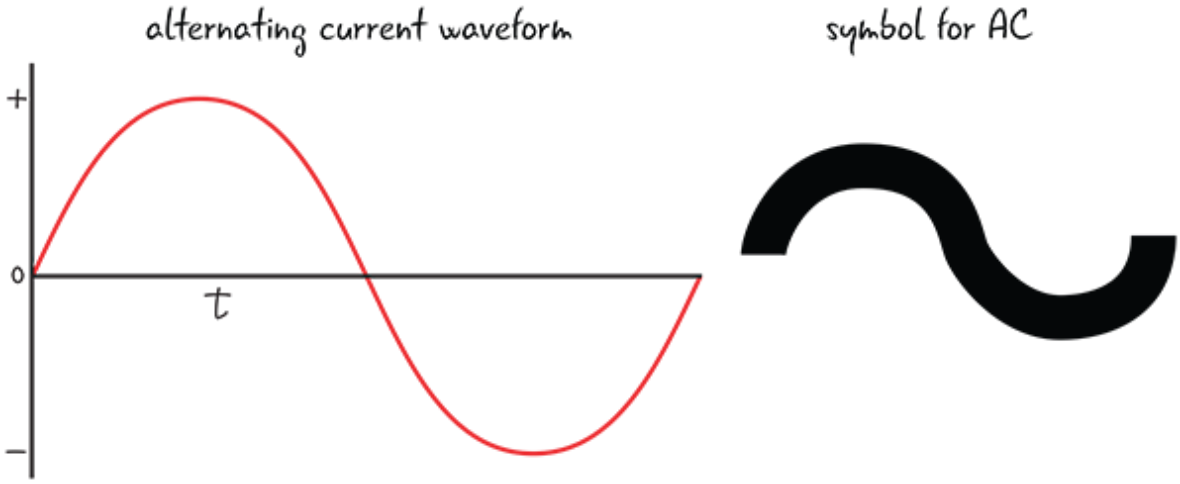


FIGURE 5-10: Alternating current

One of the major benefits of AC is that it can be distributed over great distances, something that is much more complicated with DC. AC is also able to increase the amount of voltage supplied much more efficiently than DC. Small-scale electronics projects, such as those we create with the Arduino, do not need to transport the electricity great distances, nor do they generally require large amounts of voltage. For these reasons, our descriptions of electricity will be restricted to direct current, which we describe in more depth in the following pages. Direct current, which most small-scale electronics projects use, is represented by Figure 5-11.

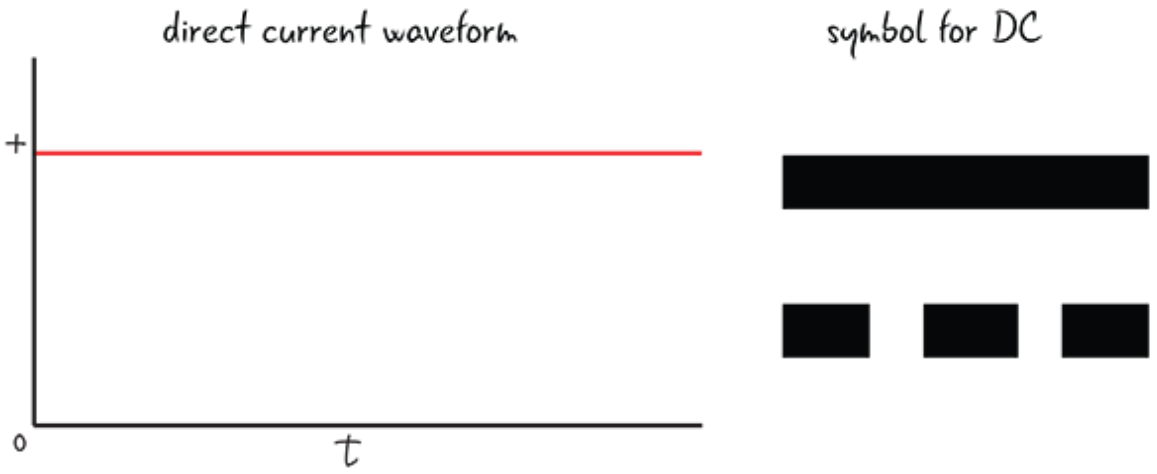


FIGURE 5-11: Direct current

Warning

Electricity is dangerous. Don't try using anything that involves alternating current!

QUESTIONS?

Q: Is a battery direct current?

A: Yes, a battery uses direct current.

Q: Is the power supply we use with the Arduino AC or DC power?

A: The power supply for the Arduino actually converts AC into DC power. It uses a transformer, which we are not going to cover in this book.

UNDERSTANDING ELECTRICITY: THE WATER TANK ANALOGY

Let's look at the three main properties of electricity: voltage, current, and resistance. We're exploring how electricity works in DC; AC works somewhat differently, and we aren't addressing it here. This chapter provides enough information about these properties to let you build your own Arduino projects; however, if you have a deeper interest in electronics and electrical engineering, you'll need to know more than the simplified overview presented here.We suggest these titles: *Getting Started in Electronics*, by Forrest M. Mims, III (Master Publishing, Inc., 2003); *Make: Electronics: Learning Through Discovery, 2nd Edition*, by Charles Platt (Maker Media, 2009); and *Practical Electronics for Inventors, 4th Edition*, by Paul Scherz and Simon Monk (McGraw-Hill Education, 2016).

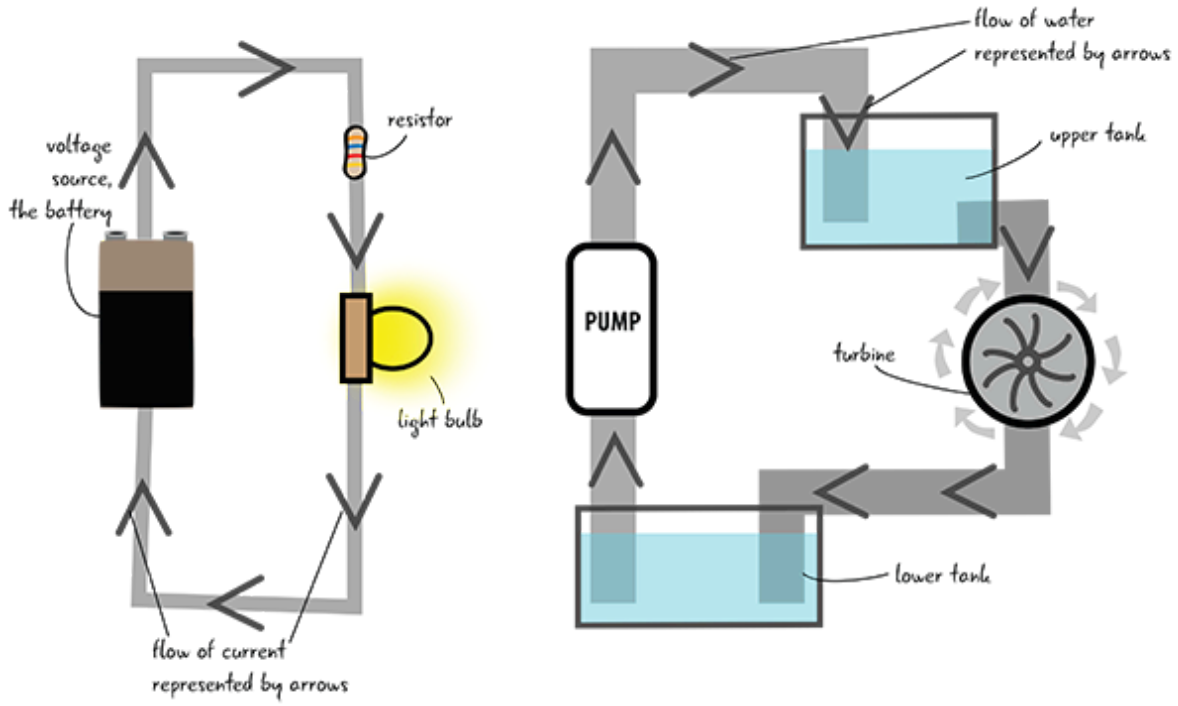


FIGURE 5-12: The water analogy for electricity

To help you comprehend how voltage, current, and resistance interact with one another, we'll use a common analogy that relates electrical properties to a water system, shown in Figure 5-12. On the left, you see an electrical circuit that has a voltage source, a light bulb, and a resistor. The flow of electricity is represented by the arrows. On the right is a water system, with a pump, two tanks of water, a turbine, and pipes connecting all of the pieces. The flow of water is represented by arrows.

QUESTIONS?

Q: Why do we use the water analogy to understand electrical properties?

A: Electrical concepts are necessarily abstract and difficult to visualize. Although the water analogy is a simplification of how electrical properties work, it helps you to conceptualize their interaction.

How do we use this analogy to understand the properties of electricity? Let's look at voltage first. In our electrical system, we have a voltage source: a battery. What is this analogous to in our water system?

VOLTAGE: THE POTENTIAL

In our water system, water has a *potential* to fall from the upper tank, moving through the turbine to the lower tank. When the water gets to the lower tank, it has no more potential to fall (because it's already at the lowest point in the system). If we increase the amount of water in the upper tank, we increase the pressure, or the potential for the water to fall. If there is increased water pressure, the turbine will turn faster, producing more work. If we decrease the amount of water in the tank, there is less pressure or potential to fall and so the turbine will turn more slowly, doing less work (Figure 5-13).

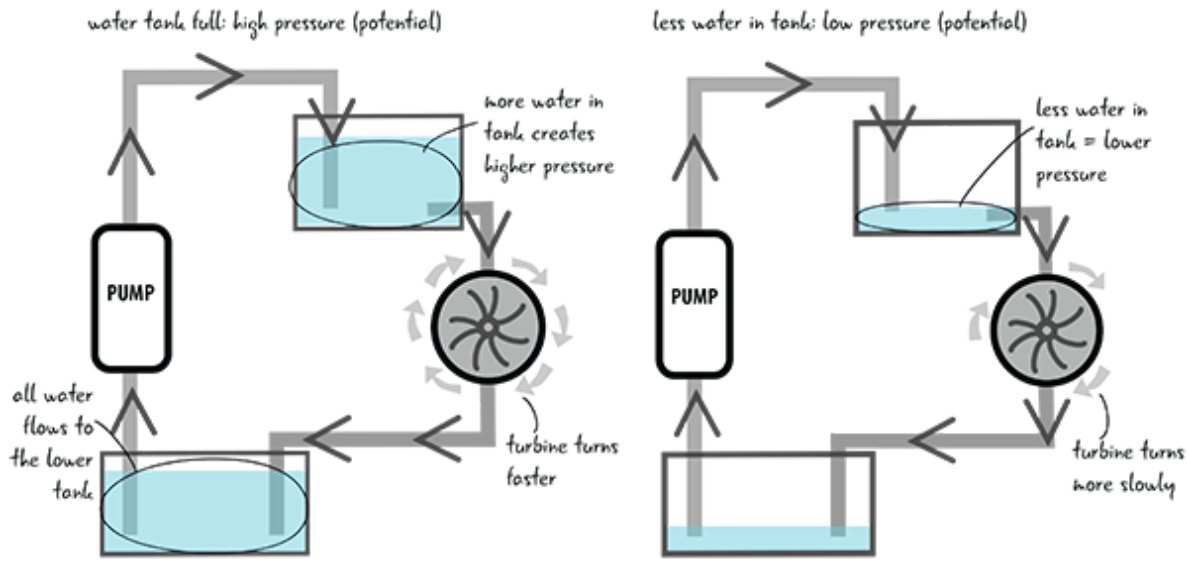
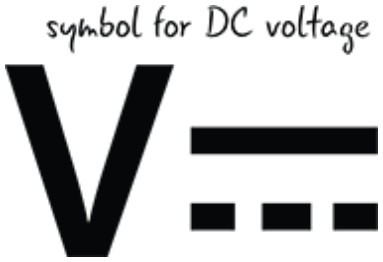


FIGURE 5-13: Voltage in the water analogy



How does this relate to voltage? In our electrical system, the electrons have the *potential*—that is, they have the pressure to flow from an area of higher charge to lower charge. Similar to more water making the turbine spin faster, in a circuit, a higher voltage source makes the light shine brighter (more electrical potential), whereas a lower voltage source (less electrical potential) makes the light dimmer, as shown in Figure 5-14. This potential is also known as the *electromotive force*.

Note

Electromotive force is the potential for electricity to flow.

Similar to the way water always flows downhill (from a higher point to a lower one), *electricity has to flow from a higher voltage point toward a lower voltage point*. Measuring voltage involves measuring the difference between the pressure at any two points in the system. It is always a *relative* value, measuring the difference between two points. Figure 5-15 shows the schematic and the electrical model, with the flow of electricity marked from power to ground.

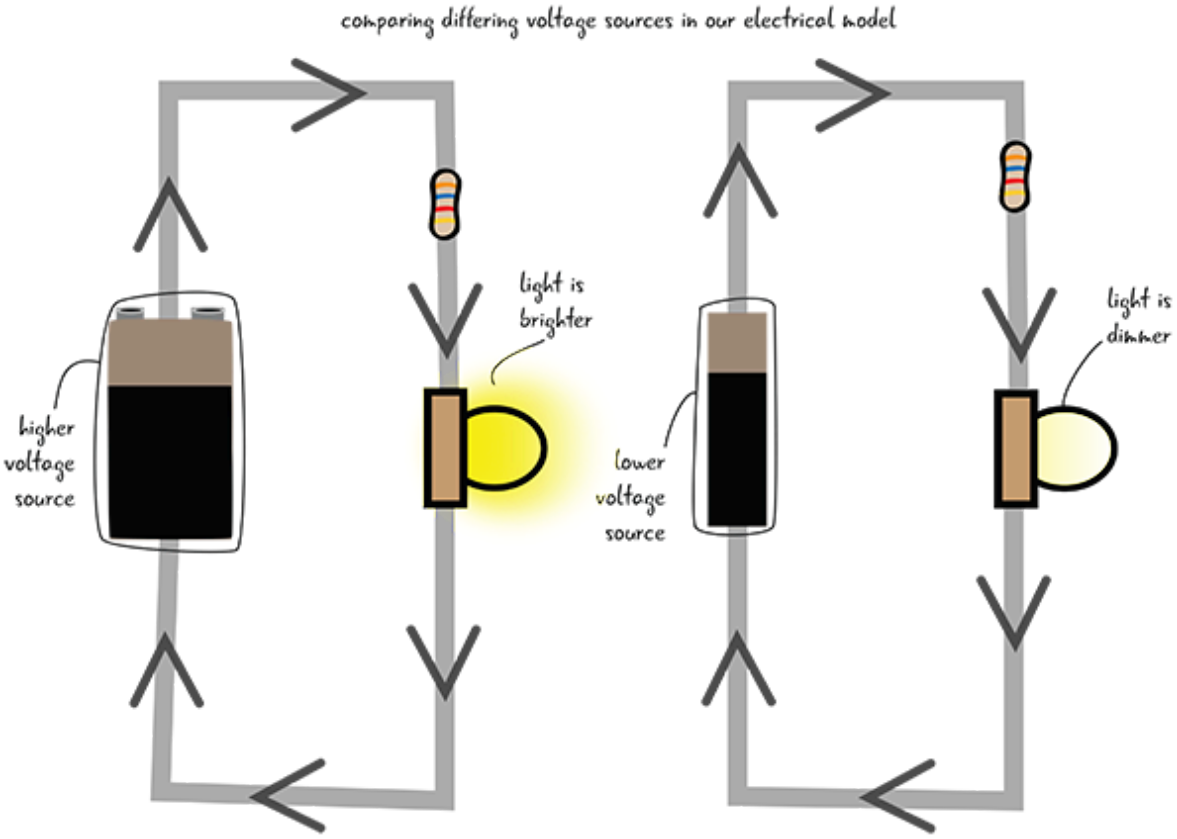


FIGURE 5-14: Electrical model with differing voltages

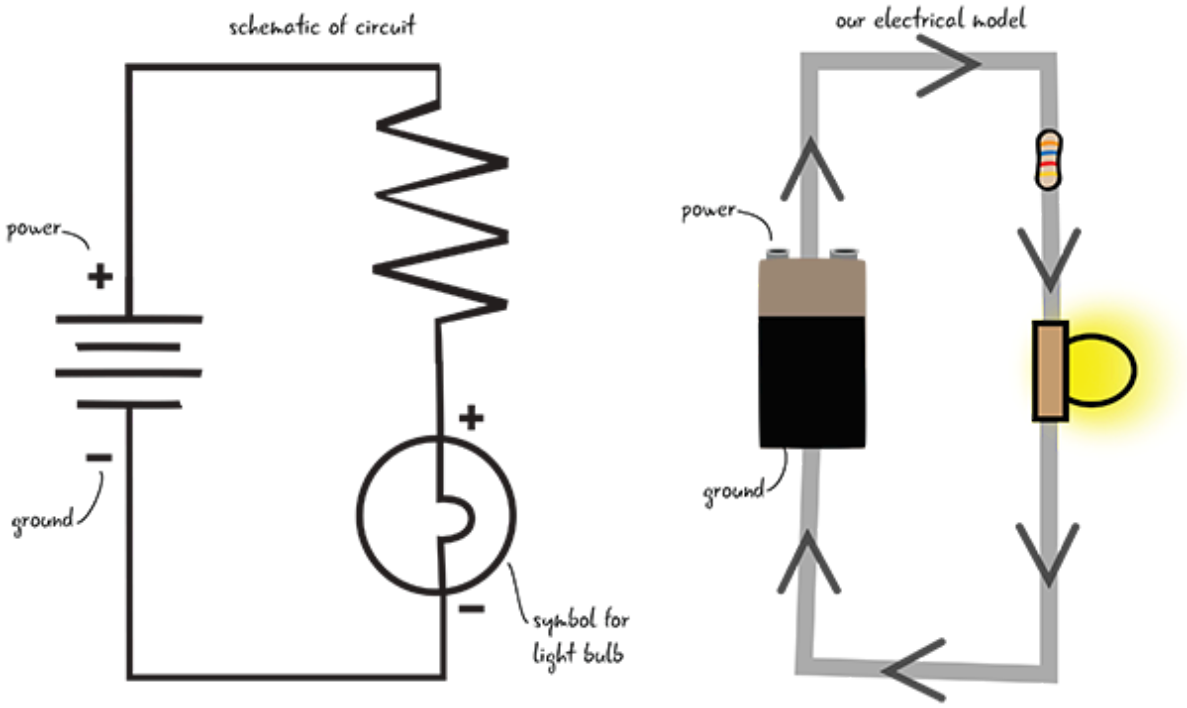


FIGURE 5-15: Our electrical model with the schematic

Note

Voltage is the difference in electrical potential between any two points in the circuit.

As the electricity travels through a circuit and its components, from a point of higher to lower voltage, this electric potential is consumed and used up by the components it flows through until there is no more potential energy. This zero point, measured at zero volts, is also known as the *electrical ground*. It is analogous to the lower tank in the water circuit, the lowest point in the system, where the water can't fall any lower—it has no more potential to fall. This is the same ground that we've talked about in our circuit diagrams and on our Arduino.

Note

Zero volts (the point of no electric potential) is known as *ground*.

QUESTIONS?

Q: So is the ground you said is zero volts the same ground we've been talking about in our circuits since Chapter 3?

A: Yes, ground is a reference point in a circuit with the electrical potential equal to zero.

WHAT'S THE VOLTAGE VALUE FOR AN ARDUINO?

You may be familiar with voltage as appliances, electronics, and electrical components all often list a voltage rating. Most small-scale electronics, like phone chargers, use between 3 and 12 volts DC.

Our Arduino operates at 5 volts. Remember how you connected the breadboard to the pin marked 5V on the Arduino? When your Arduino is plugged into your computer, it is getting 5 volts from the computer. In your circuits, the components (the LED, for example) use up some of the voltage. The resistors you've used in your circuits allow you to change (reduce) the value of the voltage, which you'll learn more about later in this chapter.

Now that you're familiar with voltage, let's see how you measure it with a multimeter.

CHECKING THE VOLTAGE

Why measure voltage? It is critical to know that your breadboard and components are receiving voltage; this is always one of the first steps to take in debugging your electronics projects.

We are continuing to use our multimeter from SparkFun (SparkFun part number TOL-12966). Back in Chapter 3, you saw that the multimeter has probes that have to be in the correct ports to measure different electrical properties. Let's check the probes to make sure they are in the right ports, and then set the dial on the multimeter to measure voltage.

Measuring Voltage

Make sure the black probe is in the COM (common) port and the red probe is in the port marked mAVΩ on the right side of the multimeter, as shown in Figure 5-16. Then turn the dial to the section that measures DC voltage. When measuring voltage, you need to set the dial on the meter to a value above what your estimated voltage is. For example, you know that the Arduino puts out 5 volts, so set the dial to 20V.

> Tip
>
> When measuring voltage, set the dial to a value greater than what you think your reading will be.

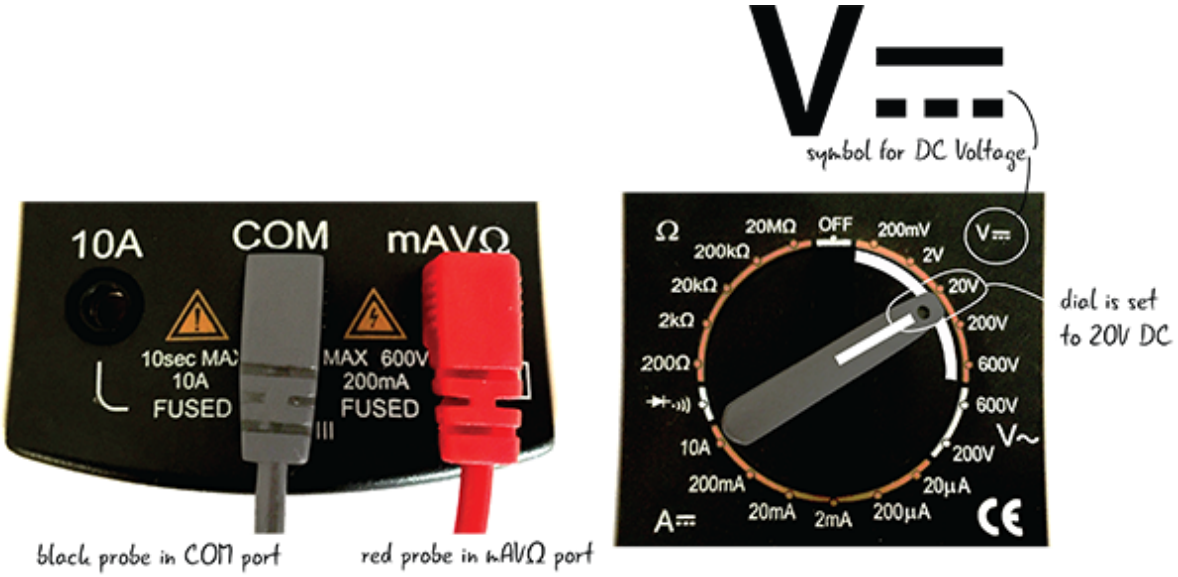


FIGURE 5-16: Settings on multimeter to measure DC voltage

We told you that 5 volts are coming out of the Arduino—let's check with the multimeter to see if that's true.

Grab a new jumper and stick one end of it into the power bus on the breadboard. The other end should not be connected to anything. Then connect a different spare jumper to the ground bus, with the other end loose. Don't let the "loose" ends of the jumpers touch each other, or you will cause a short circuit, where the path of electricity takes a shortcut to ground and potentially damages your Arduino. You can reduce the chance of a short circuit by keeping space between your two leads.

Warning

A short circuit allows electricity to travel along an unintended path; it can damage your circuit.

Next, touch the metal end of the jumper that's attached to the power bus with the red probe, and touch the metal end of the

jumper attached to the ground bus with the black probe, as shown in Figure 5-17.

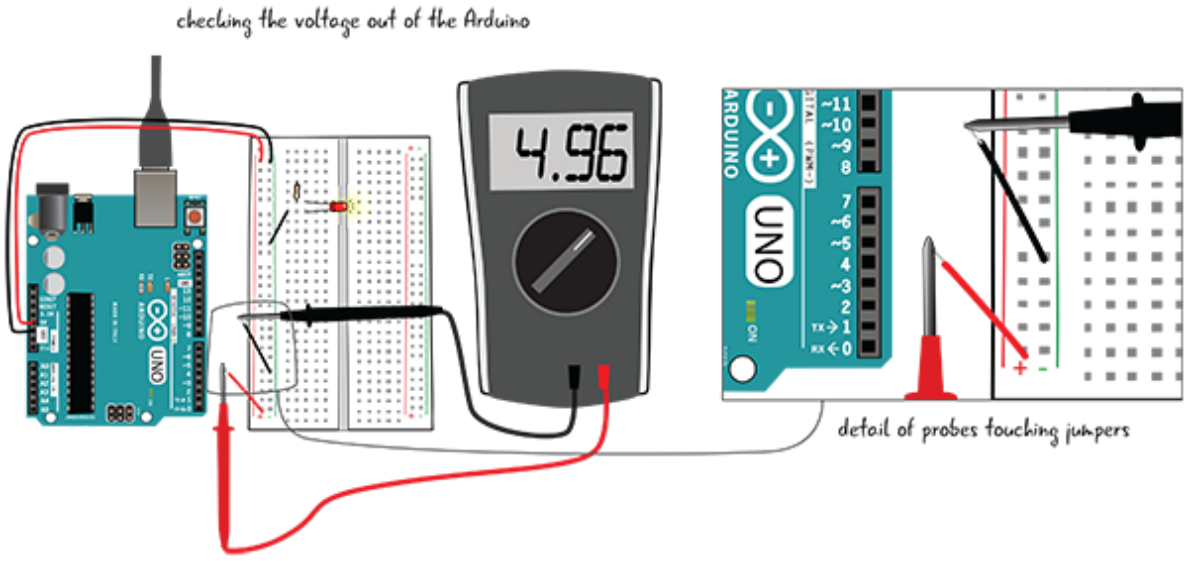


FIGURE 5-17: Metering the voltage on the breadboard from the Arduino

On the multimeter's screen, you should see the number 5, though that number may be a bit lower. Our meter read 4.96. This is the amount of voltage that comes out of the Arduino and that is going into the breadboard. The slight difference has to do with the resistance in the breadboard, the components, and/or the Arduino's internal circuitry.

What if your screen shows a negative number? That means you probably have the probes reversed; the red probe touching the jumper attached to the ground bus and the black probe attached to the power bus. Try switching the probes to the opposite jumpers and you should get a positive number.

If you see the number 1, it means that the dial on your multimeter is set to the incorrect value. Simply increase the value of the voltage dial to the next level by turning it clockwise. Then you should see an accurate value.

> **Tip**
>
> If your multimeter is showing odd values, check that your voltage dial is greater than the expected voltage. If you have a negative voltage, switch the location of your probes.

After you have measured the voltage, remove the jumpers (you don't want to cause a short circuit by letting them touch each other accidentally). We're going to measure the voltage across the components in the circuit.

> **Warning**
>
> Don't forget to turn off your multimeter when you aren't using it, or you'll run down the battery.

CHECKING VOLTAGE ACROSS THE COMPONENTS

Now you'll measure the voltage across the resistor and across the LED. Doing so will show you how much voltage each component is "using up."

Keep the dial of the meter at the spot marked 20VDC. Your Arduino should still be attached to your computer to give your Arduino power. Touch the red probe to the end of the resistor attached to the jumper to the power bus, and the black probe to the other end, as shown in Figure 5-18. What do you see on the multimeter's display?

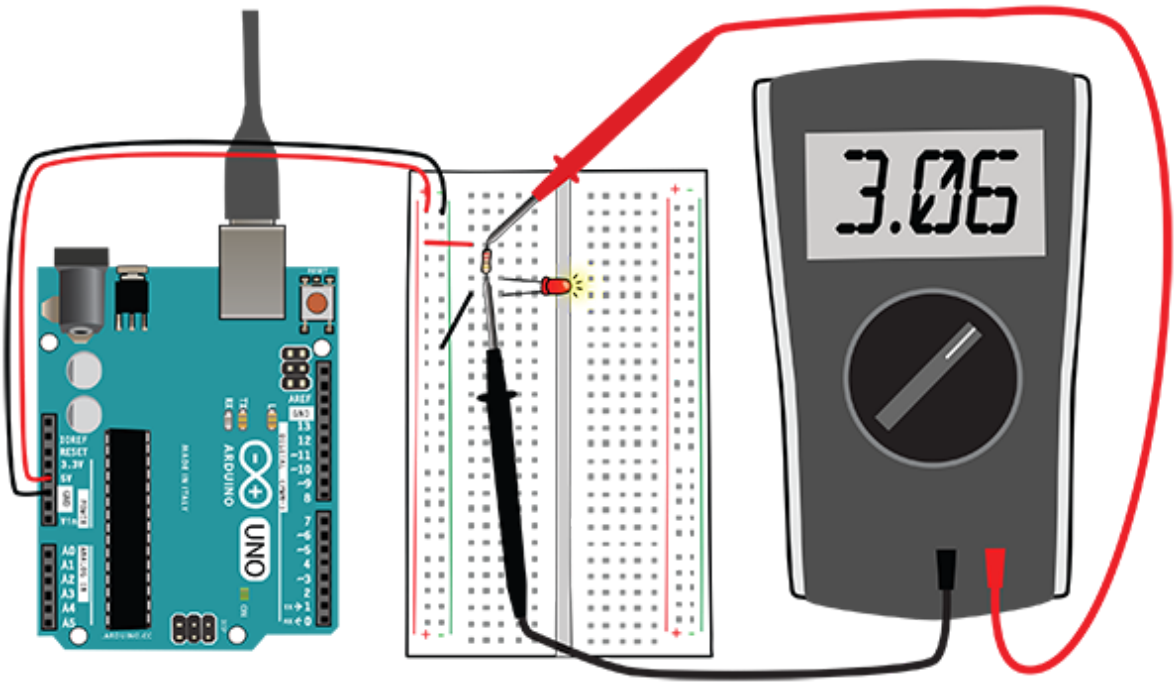


FIGURE 5-18: Measuring the voltage across the resistor

Now touch the anode of the LED with the red probe and the cathode with the black probe to see how much voltage is being consumed by the LED. Figure 5-19 shows a detail of the probes of the multimeter attached across the resistor and across the leads of the LED.

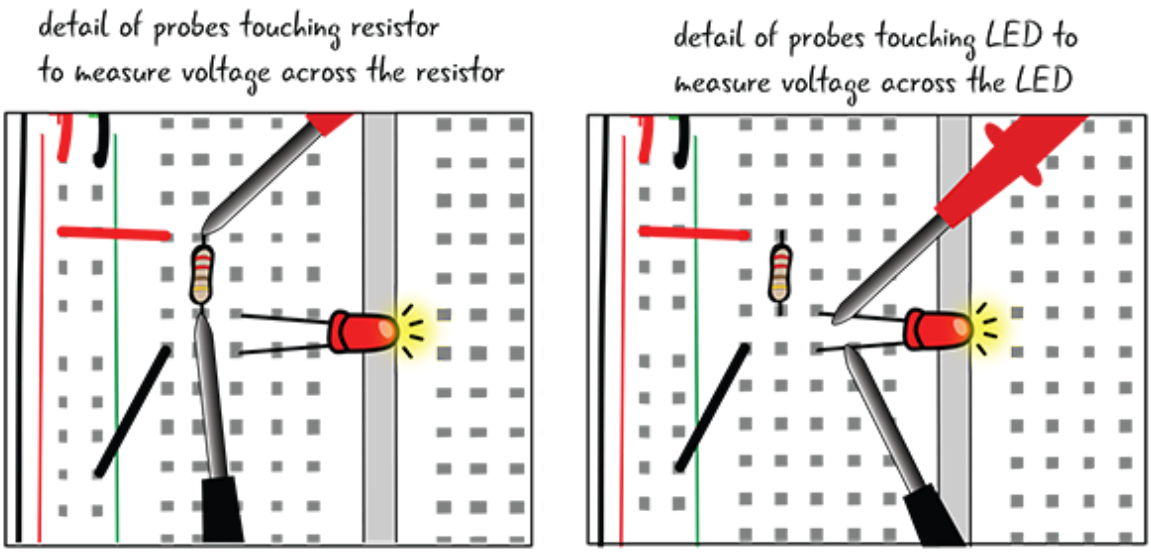


FIGURE 5-19: Details of measuring voltage in the circuit

The display on the meter should read something like 3.06 for the resistor and 1.86 for a red LED. These numbers will also vary, partially based on what color LED you're using. Don't worry that this doesn't add up exactly to 5 volts. The number displayed on the meter is the amount of voltage that the LED is using. When we measure voltage across a component like this, it is called measuring the *voltage drop*. Voltage drop is the amount of voltage consumed by a component.

VOLTAGE DROP

If voltage drop is the amount of voltage consumed by a component, what does this mean for our circuits? Each of our components will consume some of the voltage provided by our power source, until all of the voltage is consumed, as shown in Figure 5-20. If we only have one component (say, if we just placed our LED in without our resistor), then all of the voltage would flow through the one component and burn out our LED. How do we determine the amount of voltage a component consumes without damaging the component? Remember data sheets, which we discussed in Chapter 2, "Your Arduino"? They will have this information. We've already shown you

how to measure this voltage drop, but later in this chapter we'll also cover how to calculate the value ahead of time.

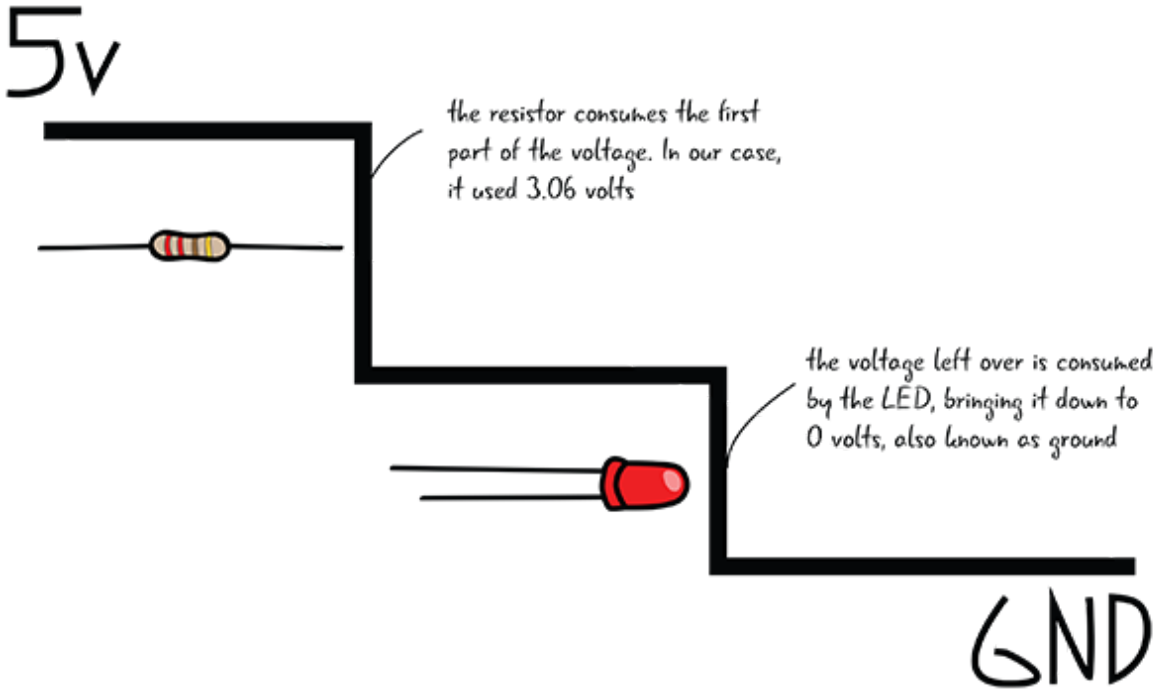


FIGURE 5-20: Visualizing the voltage drop

Note

Voltage drop is the amount of voltage consumed by a component. The components in your circuit will consume all of the provided voltage.

QUESTIONS?

Q: What are we measuring when we check the voltage across the component?

A: The number on the multimeter for voltage is the difference between the voltage going into the component and what comes out the other side. This allows us to see how much voltage the component uses.

Q: So voltage drop across a component refers to how much voltage that component is using up?

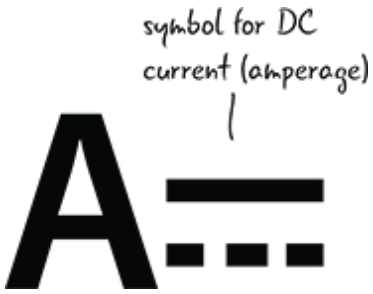
A: Yes. As you saw when we metered our components, each one is consuming some of the total voltage in the system.

Q: What happens if our components don't use up all of the voltage?

A: The components will always use up all of the voltage provided, and if a higher voltage is provided, the values they consume will scale to a higher value.

Now that you know what voltage is and how to measure it in a circuit, let's go back to the water analogy and look at current.

CURRENT: THE FLOW



In our water model, shown in Figure 5-21, we can measure the amount of water that flows through the pipes. If we measure a cross section of one of the pipes at any point, we can figure out how much water passes through it in a given amount of time—for example, we might measure 1 gallon per second. People typically refer to this as the water's current—the more water that is flowing in a given amount of time, the stronger the current.

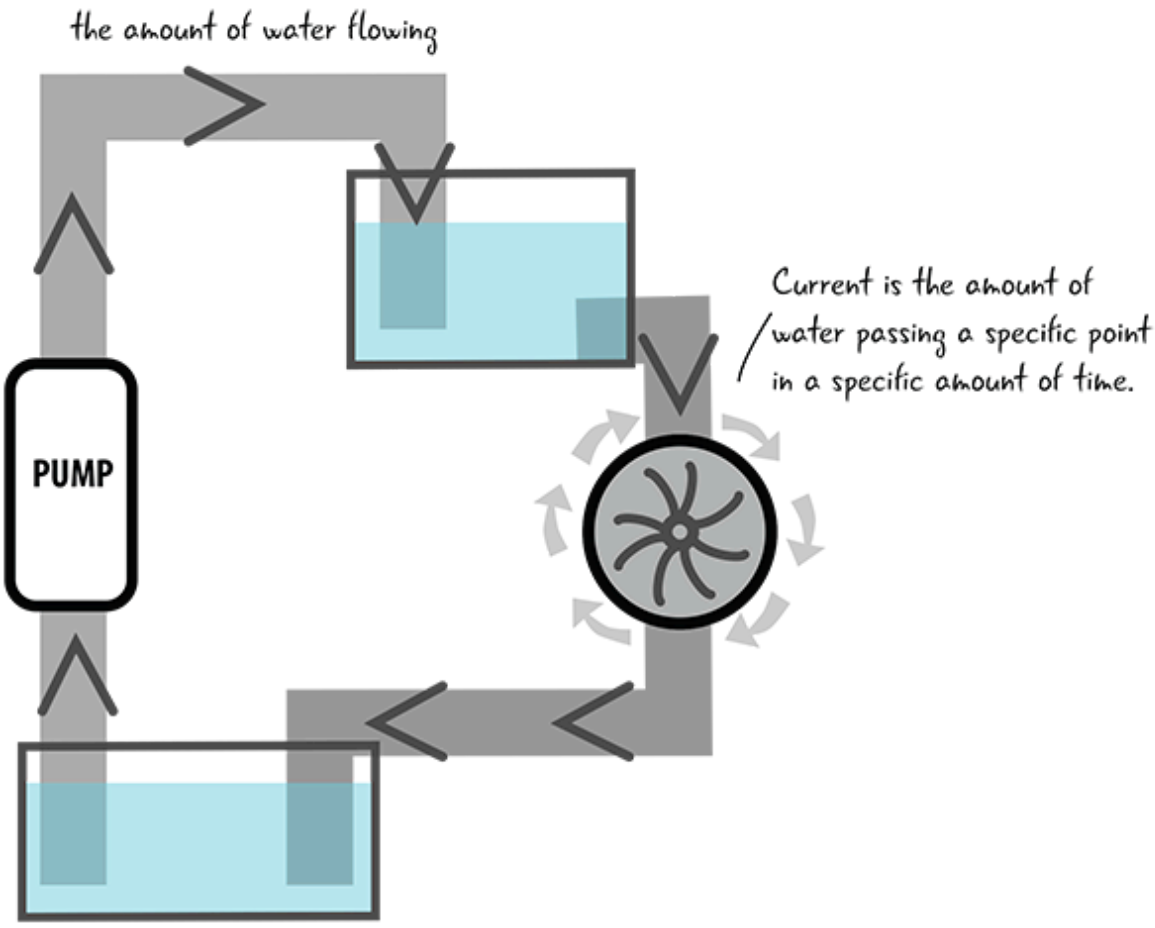


FIGURE 5-21: Current in the water analogy

The word *current* means pretty much the same thing in a circuit. Current is the amount of electrical charge passing through the circuit per second. Current is measured in amperes (a.k.a. amps), which is why current is also called amperage. Current requires a complete, closed loop in order to flow. If your circuit is not a complete closed loop (say there is a broken wire in the circuit), then there is zero current. Current in the electrical model is shown in Figure 5-22.

Note

Current is measured in amps, or amperes, which is the amount of electricity flowing per second.

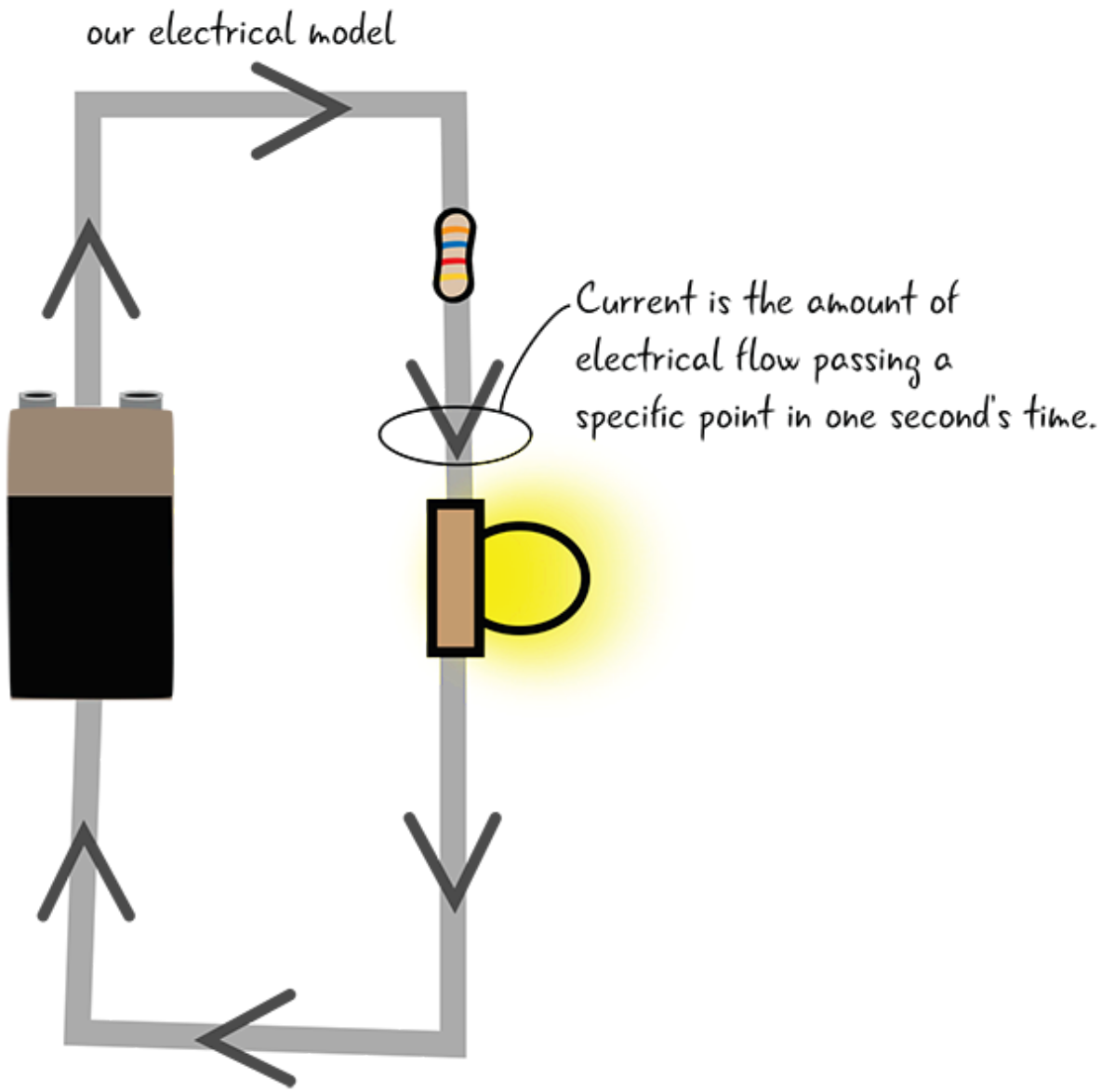


FIGURE 5-22: Current in the electrical model

CURRENT IN THE CIRCUIT

The amount of current in your circuit is determined by two things:

The Resistance of Your Components in the Circuit

Components that require more current will typically have less resistance. We'll explain more about resistance later this chapter.

The Power Supply's Current Rating

This rating indicates the maximum amount of current the power supply can produce. You can check the rating for current (as well as voltage) by looking at the *output rating* generally placed on the bottom of the power supply with other information. Figure 5-23 shows the output rating on the bottom of a power supply. (We recommended that you purchase a power supply that is rated 500 milliamps to 1 amp for current, 9–12V for voltage).

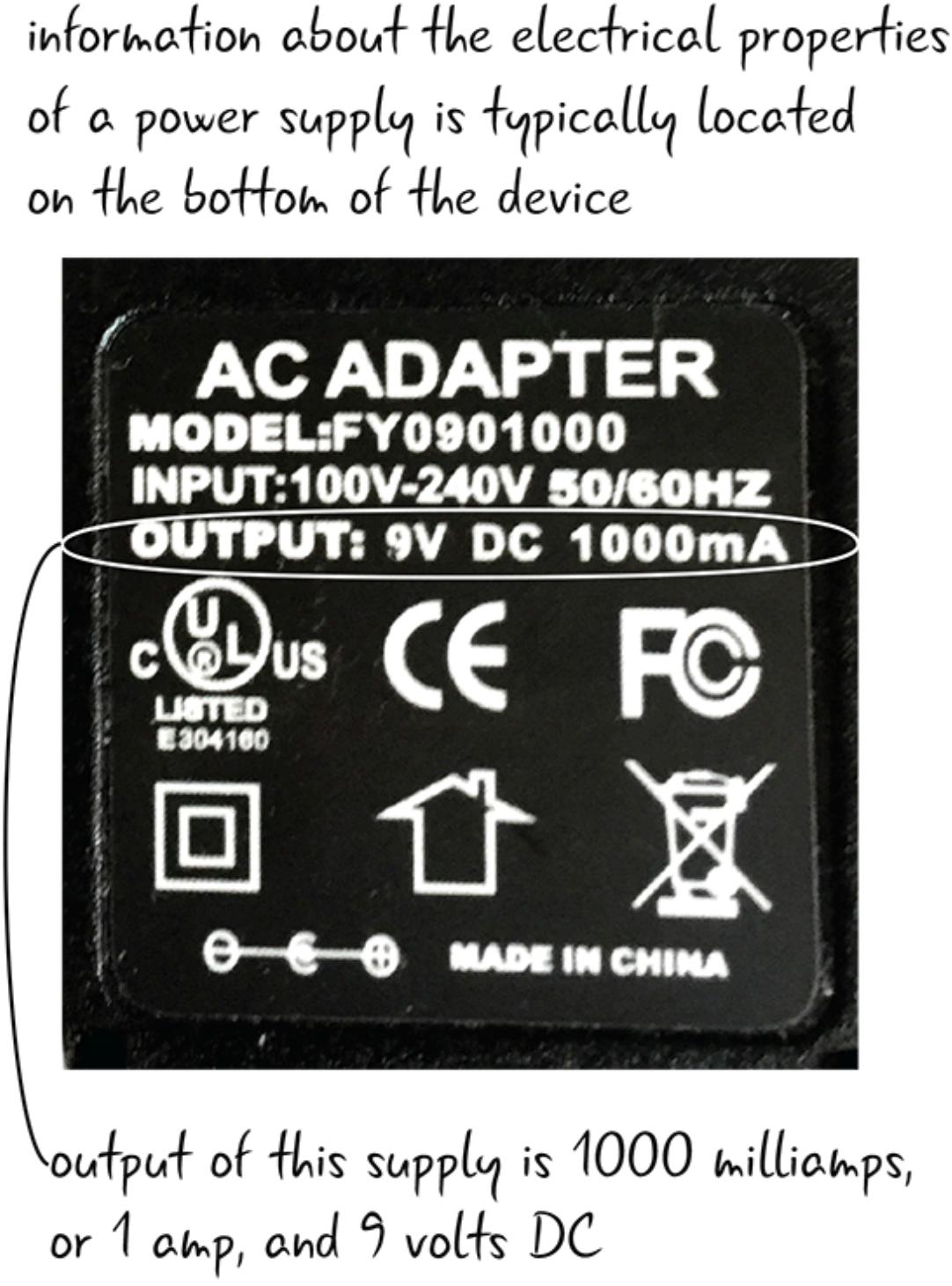


FIGURE 5-23: Output ratings on the bottom of a power supply

All components have a current rating shown on their case or listed on their data sheet, which shows how much flow they can handle. A

component can't force a power supply to push more current than the power supply is rated for.

WHAT'S THE CURRENT LIMIT FOR AN ARDUINO?

The Arduino board has a current input limit of one amp. We recommended that you purchase a power supply that is rated from 500 milliamps (½ amp) to 1000 milliamps (1 amp). The USB cord connecting your Arduino to a computer will provide 500 milliamps (½ amp), which is enough to run the Arduino board and provide power to the pins. A power supply with a higher rating than one amp could damage your Arduino.

The Arduino can only output 40 milliamps on each I/O pin. There are other electronics components that can help your Arduino cover higher current applications, but 40 milliamps is enough to power the components we'll cover in this book.

> ### Note
>
> The maximum input current for the Arduino is one amp. The I/O pins on your Arduino will only output a maximum of 40 milliamps.

Now, let's look at how current can be measured with a multimeter.

MEASURING CURRENT

Measuring current is trickier than measuring voltage, and it's done much less frequently as part of the debugging process than measuring voltage. So why are we showing you? It's a useful exercise to learn how current flows through your circuit, and to understand the difference between voltage and current. The multimeter is your primary debugging tool, so we want you to know how to use it to check many electrical properties.

To measure current, you have to pull out one of the leads of a component in a circuit, as shown in Figure 5-24, to insert your meter and make it part of the loop of the circuit. In your circuit, you'll pull out the anode of the LED. As always, when you are making adjustments to your circuit, make sure it is *not attached to power*.

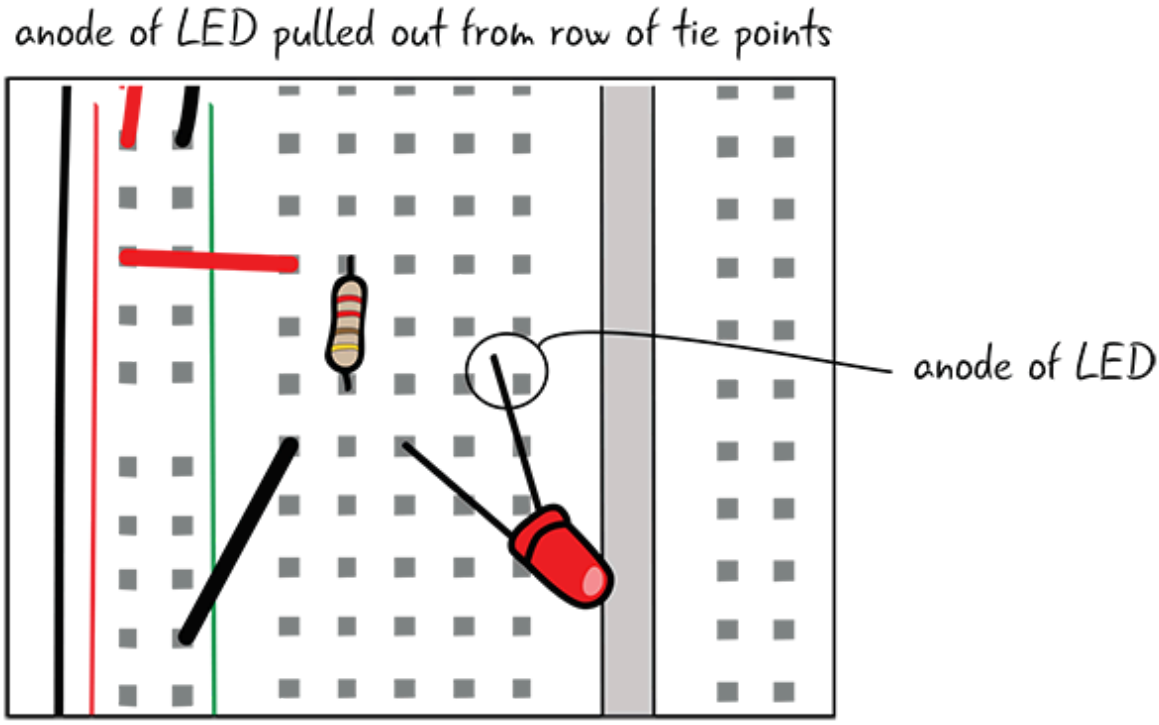


FIGURE 5-24: Pull out the anode of the LED to prepare your board for measuring current.

ADJUSTING THE MULTIMETER

You need to move the dial of the multimeter to measure 200 milliamps of DC amperage. Just like when you measure voltage, with current measurement you want to pick a value greater than what you expect the value to be—200 milliamps is the maximum safe current value for your multimeter without moving the probes (more on that later). Since you aren't using any high-current components like motors, you can feel confident that 200 milliamps will be more than the current value. So, leave the multimeter's probes plugged into the same ports, as shown in Figure 5-25.

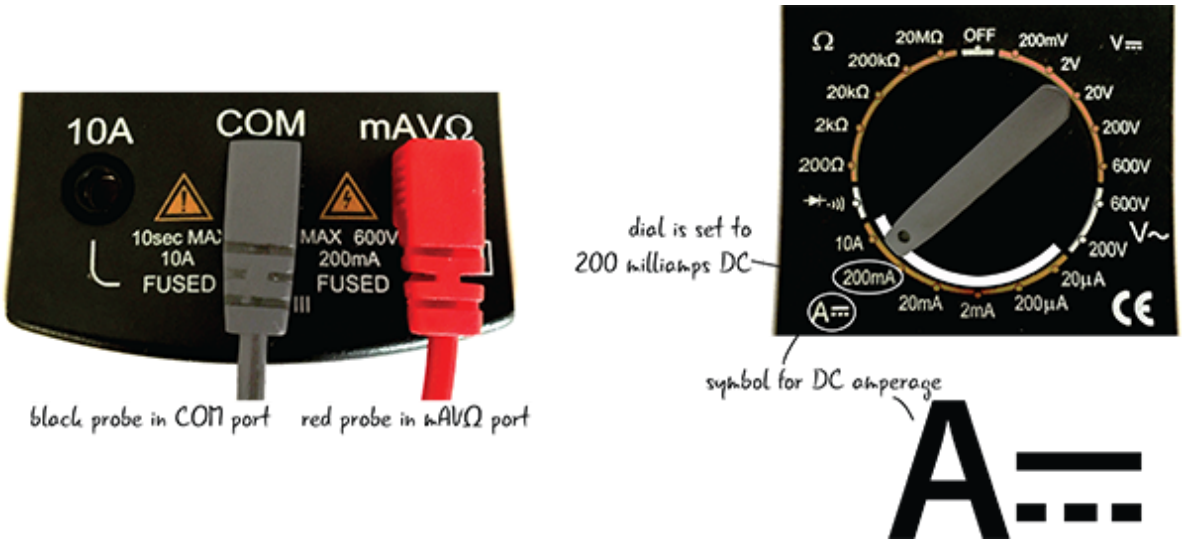


FIGURE 5-25: Settings on the multimeter for measuring small amounts of current

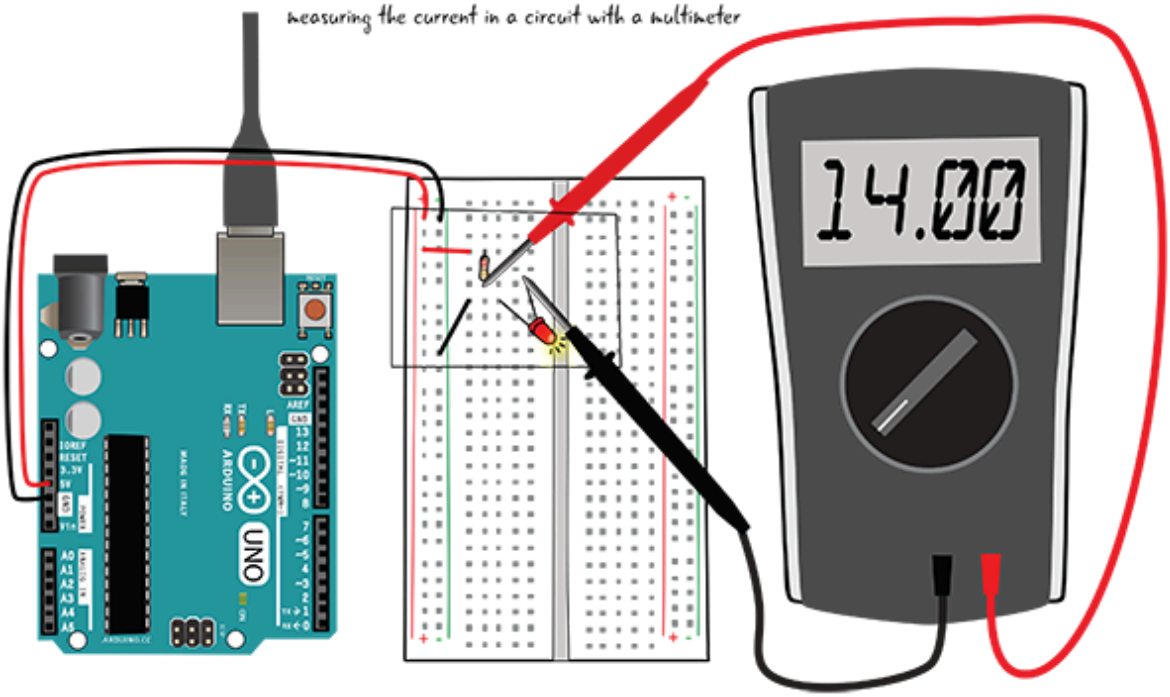


FIGURE 5-26: Probes touching one end of the resistor; the anode of LED pulled out of the tie point

Now that you've set up your multimeter correctly and arranged your circuit so that the anode of the LED is pulled out from the row of

tie points, you can plug the Arduino back into your USB cord. Next, take the red probe of your meter and touch the lead of the resistor that was in the same row of tie points as the anode of the LED (before you pulled the anode out for this exercise), and touch the black probe to the anode of the LED, as shown in Figures 5-26 and 5-27. The LED should light up because the multimeter is now a part of the closed loop of your circuit. Since the multimeter is inserted in the circuit, it displays the current (a.k.a. amperage). On our meter, it read 14 milliamps. It might read something slightly different on your meter, depending partially on the color of the LED.

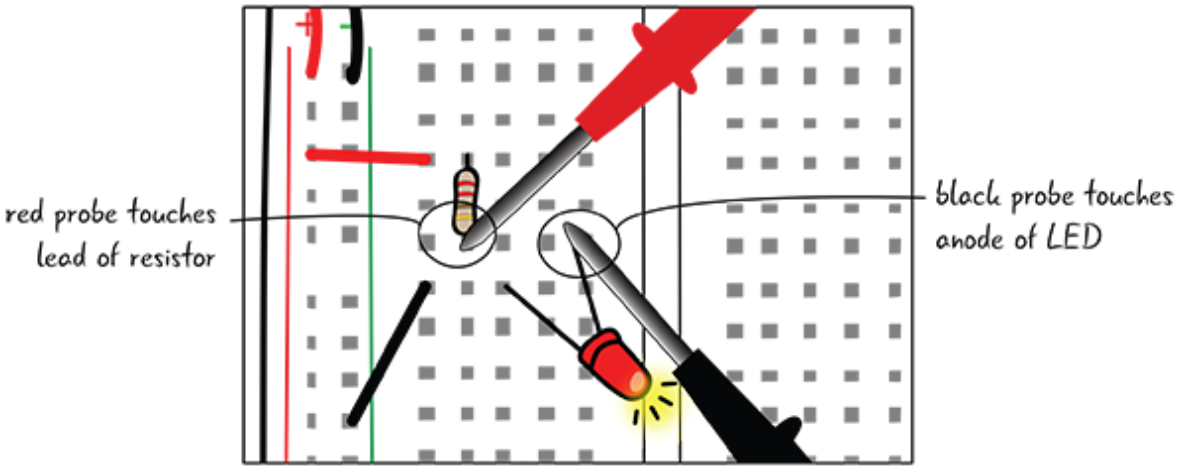


FIGURE 5-27: Detail of probe placement measuring current

Warning

Be careful when measuring high levels of amperage!

As long as you're measuring relatively small amounts of current like the 14 milliamps you measured in your circuit, it's fine to have the red probe plugged into the multimeter's mAVΩ port (milliamps, voltage, and resistance measurements). However, if you're working with stronger currents (over 200 milliamps), you need to do two things to avoid frying your multimeter:

Set the multimeter’s dial to 10A.

Move the red probe from the mAVΩ port to the 10A port.

If you forget to do these two things, the extra current can damage your meter. We recommend that you don’t measure values of current higher than 200 milliamps.

It is a good idea to keep the red probe in the mAVΩ port—that is the correct port to use for measuring most of the electrical properties.

QUESTIONS?

Q: Can we control how much current flows through our circuits?

A: Yes, the amount of current is controlled by what components you have attached within your circuit. Controlling the amount of current is an important skill for safely using more power-intensive components like motors.

Q: Are current and voltage related?

A: Yes, they are. You'll see a formula later in this chapter that explains their relationship.

Q: Why is there a separate port on the meter for high current?

A: The meter needs to use different internal electrical circuits to measure voltage and high levels of current to protect the meter from damage. Low levels of current (under 200 milliamps) won't damage the voltage-measuring circuit, but anything above that can cause issues. Switching the port is the way you change which circuit is active inside of your meter.

Q: Why does measuring current require that we remove the legs of our components from the circuit?

A: In order to measure current, the multimeter needs to become a component in the circuit. All of the current in your circuit then flows through the meter so it can figure out the total amount. We'll explain the relationship between the meter and your components a bit later in this chapter.

RESISTANCE: RESTRICTING THE FLOW

Let's look at how resistance might be demonstrated with the water analogy in Figure 5-28. If the pipes are wider in our water system, more water can flow through them. If the pipes are narrower, less water can flow. You could say that the amount of resistance, or the restriction of flow, is greater in the narrower pipes. Where there is more resistance in the system with the narrower pipes, the turbine would turn more slowly and do less work.

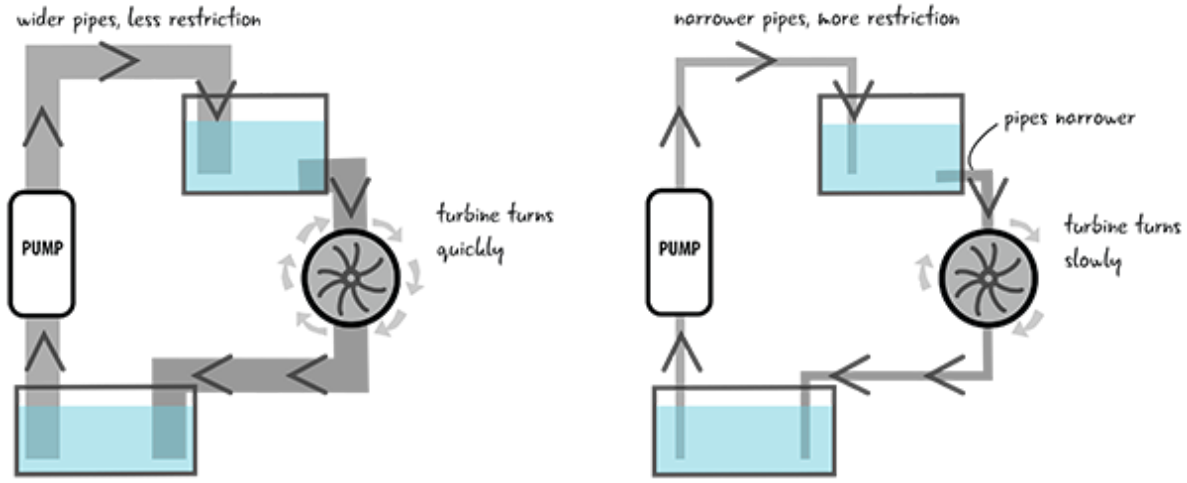


FIGURE 5-28: Resistance in the water model

Note

In circuits, *resistance* refers to how much a material restricts the flow of electricity.

In a circuit, resistors are equivalent to narrow pipes because they restrict the flow of electrons. In the electrical system diagrams in Figure 5-29, the image on the left has only one resistor and so the light shines brightly. There are three resistors on the right image, which causes more resistance value and makes the light shine less brightly.

Ohm symbol for resistance

Resistance is measured in ohms, represented by the omega symbol shown to the left. We'll look at how ohms are related to the other electrical properties later in this chapter, but at the moment just know that a resistor has a value that indicates how well it opposes the flow of electricity.

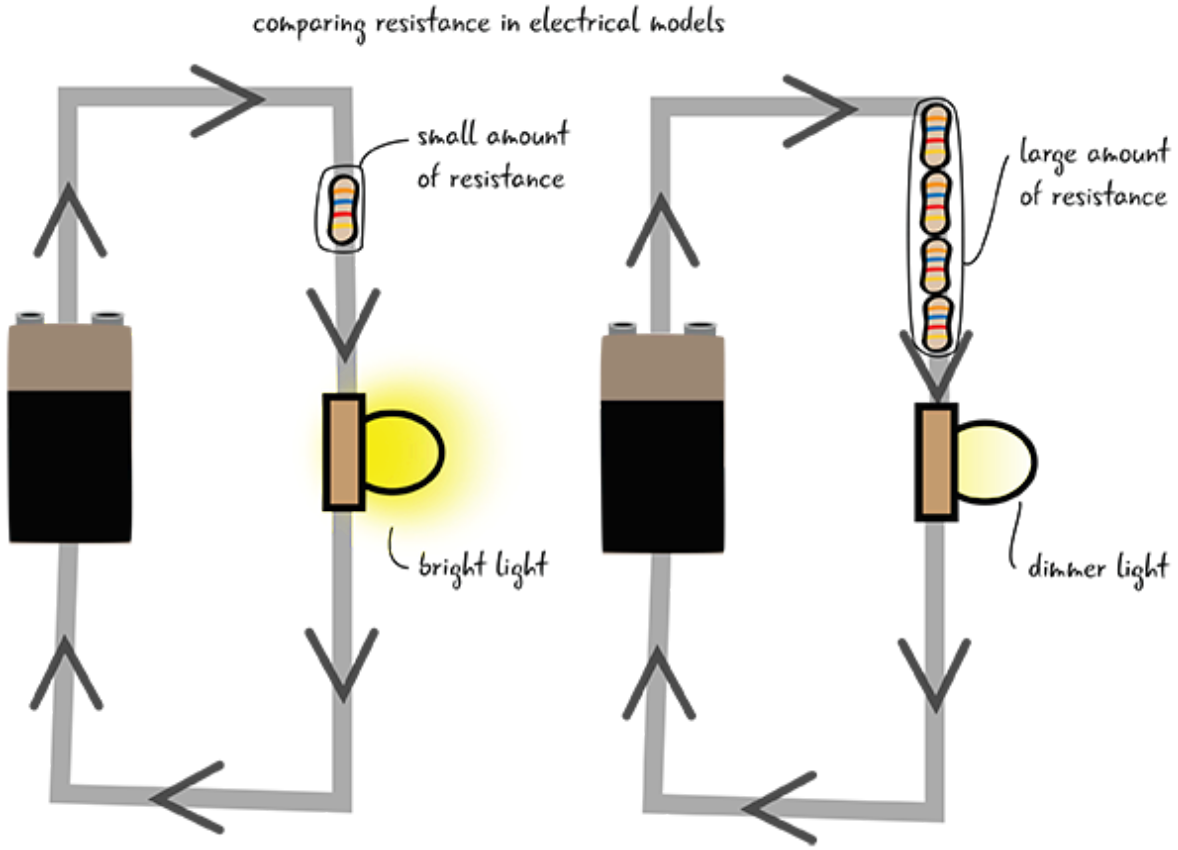


FIGURE 5-29: Resistance in the electrical model

RESISTORS UP CLOSE

As you've seen throughout the chapter, the voltage and current within your circuits vary based on what components make up the circuit. Electronic components can be very sensitive to spikes in electricity. Also, if you have a voltage source that is too powerful for a component, it could damage the component. How can you protect your electronic components within circuits? The answer is resistors. Figure 5-30 shows a package of 220-ohm resistors. You've already been using resistors to protect your LEDs from the 5V power coming from the Arduino.

FIGURE 5-30: A package of resistors

If resistance is a property of all electronic components, why do we need a special resistor component? Resistors are great because they come in a wide range of different values and can help control the

flow of electricity in a circuit. You've already used circuits that require 220-ohm resistors, but circuits throughout the book will need resistors with different values. How will you be able to identify how much resistance any given resistor has? There are a couple of ways. Let's look at measuring resistance with a multimeter.

MEASURING RESISTANCE WITH A MULTIMETER

You measure resistance in a resistor outside of a circuit. This is different from what you've seen when measuring voltage or current, where you measured these values within a circuit. Now you're going to measure your 220-ohm resistor.

On your multimeter, the black probe should be in the COM port, and the red probe should be in the port marked mAVΩ.

Move the dial so it is in the section that measures resistance. You'll set the dial to 2KΩ for this example. The correct configuration is shown in Figure 5-31.

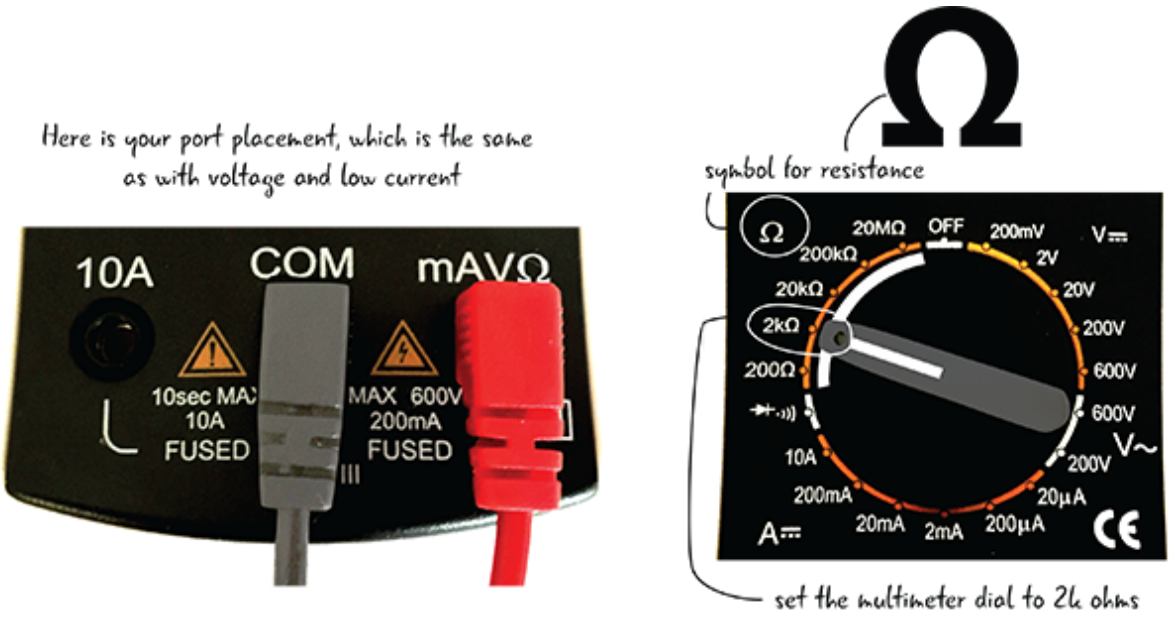


FIGURE 5-31: Multimeter settings to measure resistance

You learned about setting the range when you were measuring voltage. You need to set the range when you're measuring resistance

as well. You know your resistor is 220 ohms, so you must set the dial to a value that is greater than that—the 200 ohms setting will be too low. Move the dial to 2kΩ; you're looking for a value between 200 ohms and 2000 (2k) ohms. Now that you've set the dial and you know the probes are in the right place, you're ready to measure your resistor.

Touch one probe from the multimeter to each of the metal legs of the resistor, as shown in Figure 5-32. When you're measuring resistance, it doesn't matter what side each probe is on. You may have to hold the resistor's leads so that they have a solid contact point with the probes, or you can set the resistor flat on a table. What value does your multimeter display? The display should show something close to .221, which is measured in kilo-ohms. Remember, since the meter is set to measure 2k ohms, or 2000 ohms, .221k ohms is actually equal to 221 ohms.

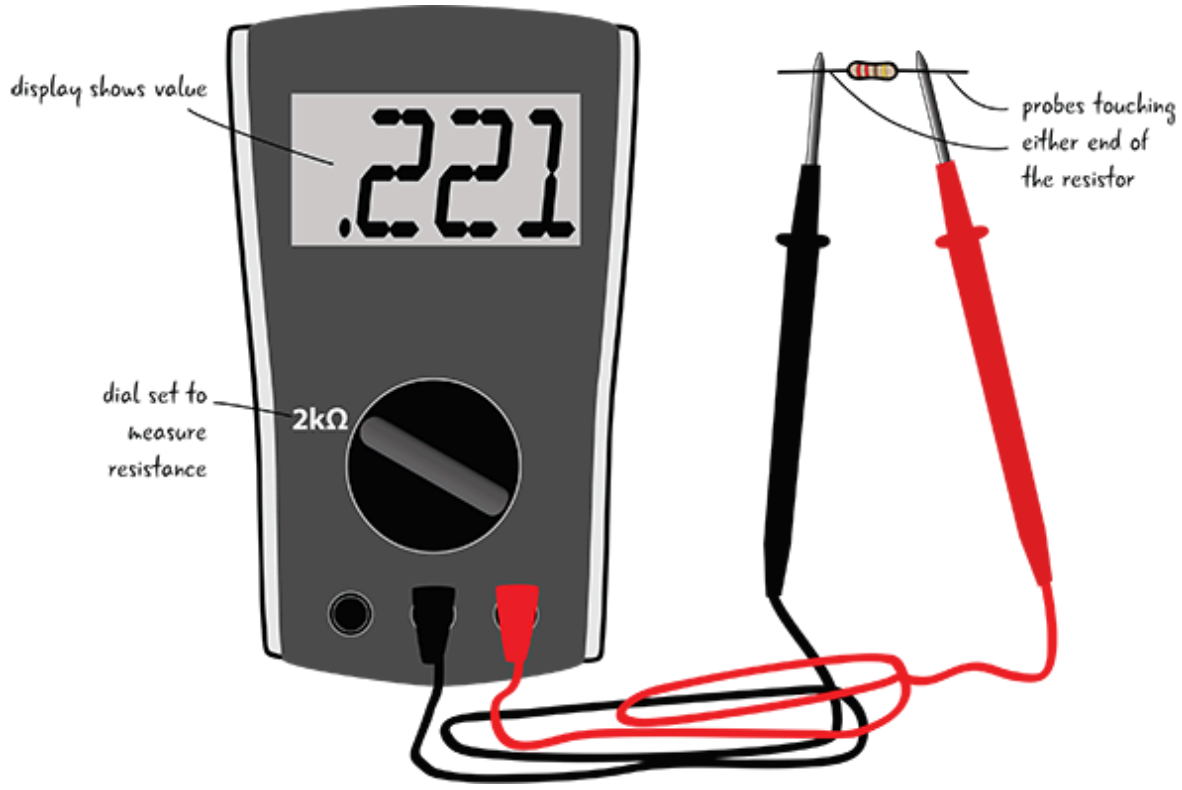


FIGURE 5-32: Measuring resistance with the multimeter

The value of the resistor will be shown on the meter display. Figure 5-33 shows what it looked like on our meter.

FIGURE 5-33: Multimeter display showing resistance value

Why is the value slightly different from the 220 ohms the resistor is rated at? It's because resistors have a tolerance value, which tells you the accuracy range of the resistor. The resistors you'll deal with in Arduino projects can have actual values that are plus or minus 10 percent different from the stated value. Generally speaking, you're working with components that aren't sensitive enough to be bothered by these discrepancies, so you don't need to worry about the variation.

Resistors include a set of color bands to help you identify their value and their accuracy. The appendix explains how to read these color bands.

QUESTIONS?

Q: What do voltage, current, and resistance have to do with the Arduino?

A: When you are working with the Arduino, you are building circuits that use electricity. If you understand how voltage, current, and resistance work, it will help you debug your circuits and eventually build more complex projects.

VOLTAGE, CURRENT, RESISTANCE: REVIEW

Let's look at our water analogy diagram one last time (Figure 5-34), then quickly review the properties you've just learned about, what unit each is measured in, and the symbol used to represent it.

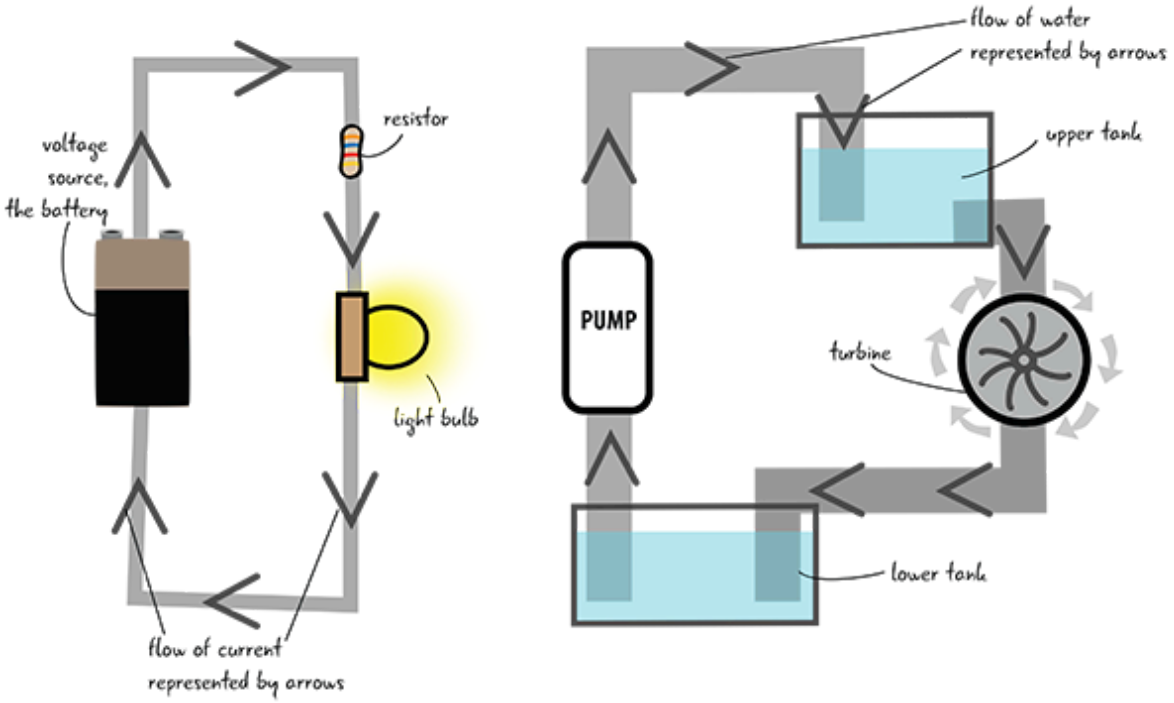


FIGURE 5-34: Water analogy for electricity

Table 5-1 reviews the electrical properties with their symbols and the units that they are measured in.

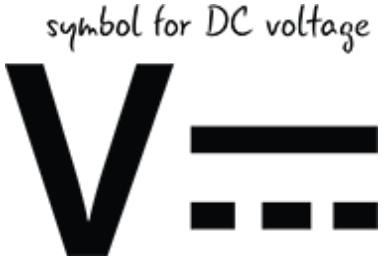
TABLE 5-1: Electrical properties

NAME	DESCRIPTION	UNIT	SYMBOL
Voltage	Electromotive force, or the potential flow	Volt	V
Current	Amount of electrical flow	Ampere, or amp	A
Resistance	Restriction of electrical flow	Ohm	Ω

HOW DOES ELECTRICITY AFFECT OUR COMPONENTS?

In a circuit, current, voltage, and resistance are related. If you have current in a system, then there is necessarily a voltage and a resistance. Let's examine what happens when you reduce only one of these properties.

Voltage



Remember that voltage represents the potential for electricity to move within a circuit. Voltage will always flow from the highest to lowest charge until it reaches the equilibrium zero state, also known as ground. If we place the same LED and resistor in our circuit, and power it using only 3.3 volts instead of the 5 volts we have been accustomed to, then our LED will be less bright. If we continue to

reduce the voltage, our LED will continue to dim until it finally turns off.

Current

symbol for DC amperage (current)

Current is the property related to the flow of electrons in the circuit. Current is what drives our components. What happens if we don't have enough current within our circuit? Without enough current there are not enough electrons to turn our components on. When you have a flashlight with dead batteries, the batteries have too little current to turn the light on. If we reduce the current to our circuit by adding resistors, the LED will turn off suddenly once the minimum current needed to turn the LED on disappears.

Resistance

Ohm symbol for resistance

Resistance is a measure of how a material restricts the flow of electricity. All materials naturally have some resistance, but if the resistance is too high the electrical flow will be stopped altogether. However, if there is too little resistance our components can be overwhelmed by the amount of current and fry. We often use resistors to restrict the flow in order to preserve the other components of our circuits. If we add more resistors or change the

value of the resistors in the basic circuit with the LED, we will increase resistance value and decrease the amount of electricity that reaches our LED, perhaps even limiting our LED's ability to produce any light.

HOW ARE OUR COMPONENTS AFFECTED BY A CHANGE IN ELECTRICAL PROPERTIES?

Let's take a quick look at how our components are affected by changes in electrical properties in Table 5-2.

TABLE 5-2: Effects of changes in electrical properties on components

PART	IMAGE	VOLTAGE	CURRENT	RESISTANCE
LED		The LED will get dimmer as the voltage gets lower, or brighter as more voltage is added; if there is too much voltage the LED will burn out.	LEDs need only a very small amount of current to run. However, reducing the amount of current too much will turn off the LED.	LEDs have a tiny amount of resistance.
Resistor		Voltage is converted into heat when it crosses over a resistor. More voltage means more heat	Resistors lower the amount of current being drawn in a circuit.	The amount of resistance depends on the resistor's rated value. Check the appendix to learn how to identify resistor values by color.

PART	IMAGE	VOLTAGE	CURRENT	RESISTANCE
		and less voltage means less heat.		
Battery		Batteries establish the voltage level for both the high point and zero volts, a.k.a. the ground.	Current comes from the battery. The current flowing will change depending on what components are attached to the battery and how much current they require.	Since a battery is not a perfect conductor, there is a small amount of resistance inside of the battery, but when it is in our circuits it is effectively zero.

Now let's take a look at how voltage, current, and resistance interact with each other in a rule called Ohm's law.

HOW DO VOLTAGE, CURRENT, AND RESISTANCE INTERACT? OHM'S LAW

Voltage, current, and resistance are related through a formula known as Ohm's law. Ohm's law, shown in Figure 5-35, states that in a given circuit, the voltage (in volts) is equal to the current (in amps) times the resistance (in ohms).

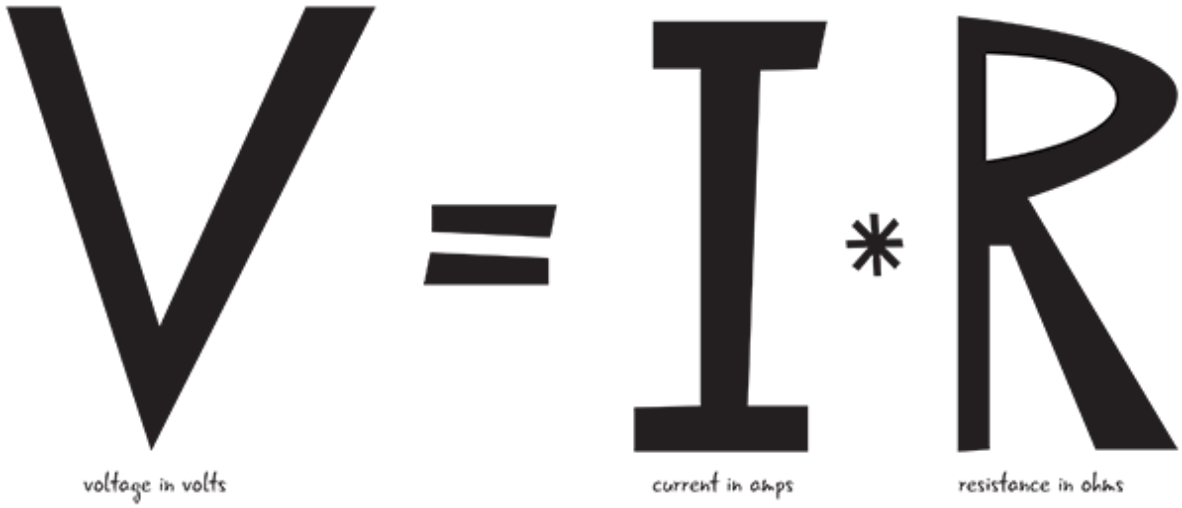


FIGURE 5-35: Ohm's law

This equation shows us that, no matter how much pressure (voltage) there is, if the resistance is high the current will be restricted. This is true for all electrical wiring.

One benefit of Ohm's law is that if we know two of the electrical properties, we can always calculate the value of the third property. You can see these relationships in Figure 5-36.

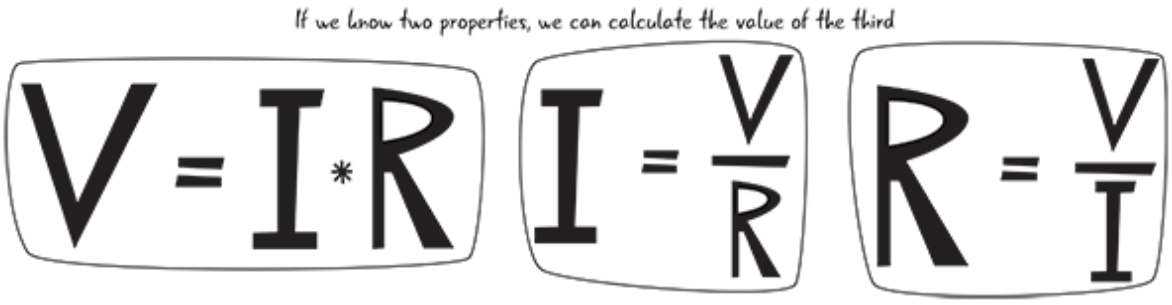


FIGURE 5-36: Permutations of Ohm's law

Since you have already learned how to measure the values of voltage and current in your circuits, you can calculate the voltage, current, or resistance, as long as you know the value for two out of the three properties.

OHM'S LAW IN A CIRCUIT

Now you know about Ohm's law, but how will it help you make your circuits? You can use Ohm's law to determine the value of the resistors you need in your circuits. You can also use Ohm's law as a safety check to confirm that the values of voltage and current running through are components are below the limits for those components.

For example, if you have a resistor in your circuit that has 220 ohms of resistance, and there is 20 mA (which is the same as 0.020 amps) running through the circuit, then you can use Ohm's law to figure out how much voltage will pass through the resistor. Figure 5-37 shows the calculations.

$$V = IR$$

$$V = (0.020\ amps) * 220\ ohms$$

$$V = 4.4\ Volts$$

FIGURE 5-37: Using Ohm's law

OHM'S LAW APPLIED

How else can you use Ohm's law? Let's say you want to build two circuits that each contain one LED and one resistor. You're going to power one circuit with a 3.3-volt pin on the Arduino, and another with the 5-volt pin (recall that the Arduino can provide either voltage). The LEDs we're going to use in the circuits take 2.2 volts to light up fully, and use 25 milliamps, or 0.025 amps. Because of the voltage difference between the two circuits, you will need different resistors in each circuit to protect the LEDs. What resistor value will you need for each circuit?

Since you know that 2.2 volts are going to pass through the LED in each circuit, you can take the difference between your provided voltages (3.3 volts and 5 volts) and 2.2 volts to figure out how much voltage will pass through each resistor (Figure 5-38).

FIGURE 5-38: Determining voltage

Now you can use Ohm's law to calculate how much resistance you need to have the stated voltage and a current of 0.025 amps pass through the resistor protecting your LED (Figure 5-39).

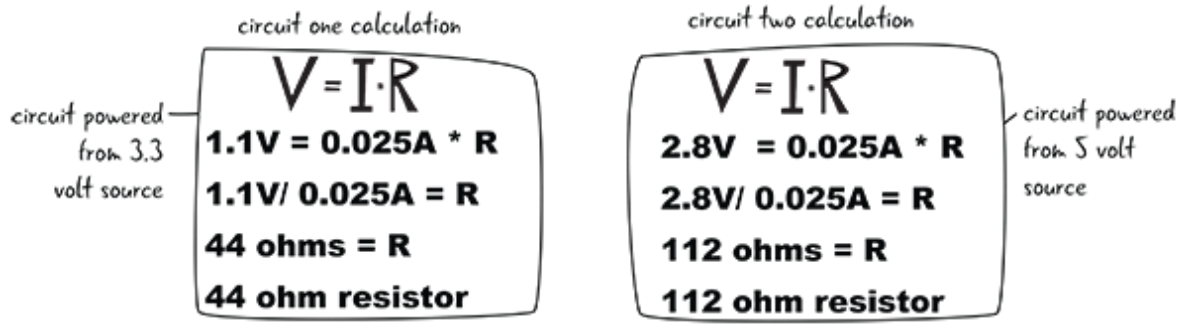


FIGURE 5-39: Calculating with Ohm's law

The 5-volt circuit requires a higher value resistor than the 3.3-volt circuit. You can see that using Ohm's law shows you how the value of the resistor required changes in your circuit based on the voltage provided. Ohm's law is useful for making sure your components are provided with the right amount of electricity.

ARRANGING COMPONENTS IN A CIRCUIT

How do you know how to arrange the components in your circuit? You know that the circuit must form a complete loop. Some components seem to be arranged next to each other with common electrical points, whereas others are connected end to end. What are these arrangements, and what effect do they have on the electrical properties in the circuit?

COMPONENTS IN PARALLEL AND SERIES

Let's look at the order of arrangement of components in a circuit. We'll look at parallel first.

ORDER OF COMPONENTS IN A CIRCUIT: PARALLEL

Components in *parallel* are placed next to each other and share electric contact points, as shown in Figure 5-40. The electricity flows along each path through the components that are arranged in parallel.

ORDER OF COMPONENTS IN A CIRCUIT: SERIES

In contrast, components in *series* follow one after another, as shown in Figure 5-41. The circuits you have built so far have all been arranged in series—all of the electricity that flowed through the resistor then went into the LED.

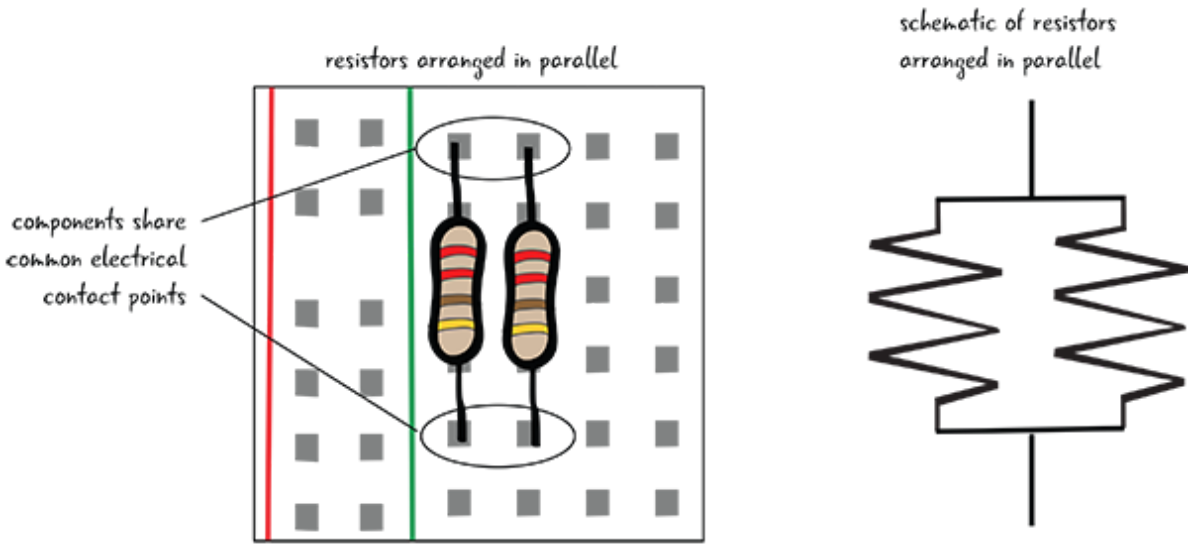


FIGURE 5-40: Resistors arranged in parallel

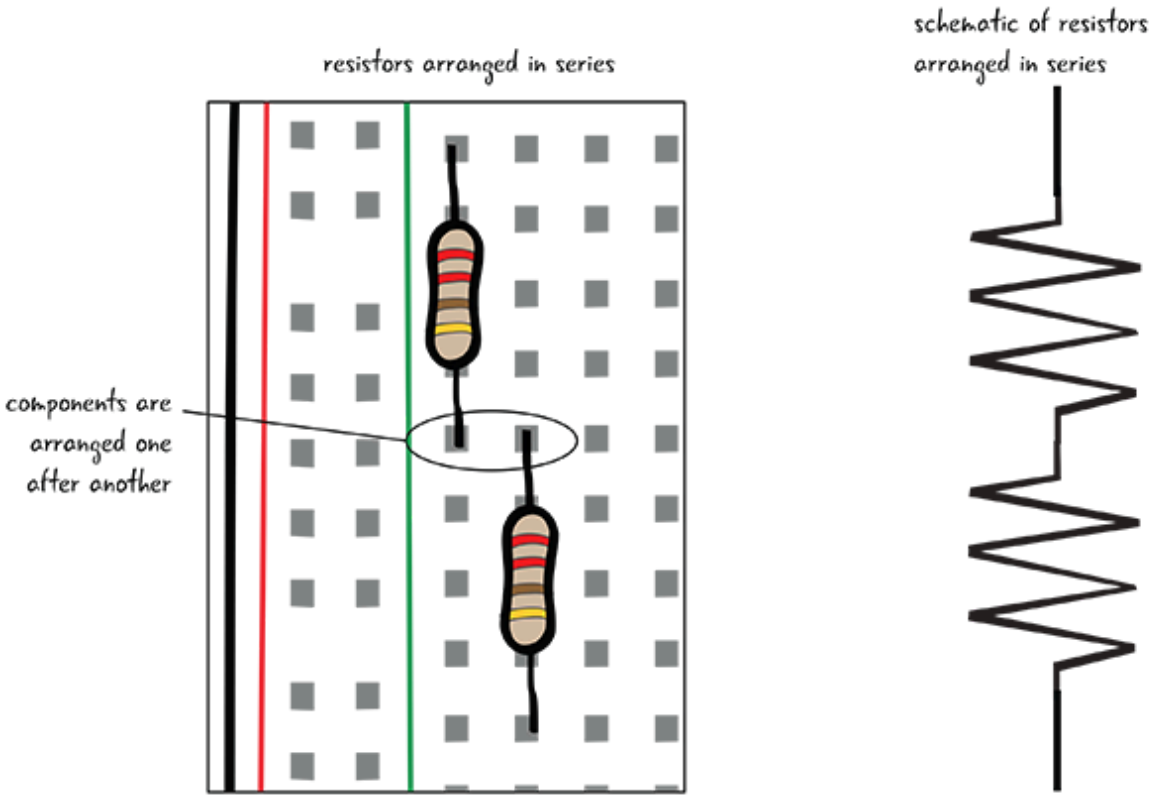


FIGURE 5-41: Resistors arranged in series

To see exactly what we mean by components in series and parallel, we'll show you how to add another LED to your basic circuit, first in parallel, then in series. Then you'll measure the voltage drop across each of the LEDs.

A CIRCUIT WITH TWO LEDS IN PARALLEL

You'll add this LED so it is in parallel with the first LED, as shown in Figure 5-42. Arranging components in parallel means that the components are connected with common electrical points. You can think of the components as being next to each other. Let's look at the schematic for a circuit with the LEDs arranged in parallel. You can see that the resistor is attached to 5 volts and also to both of the LEDs.

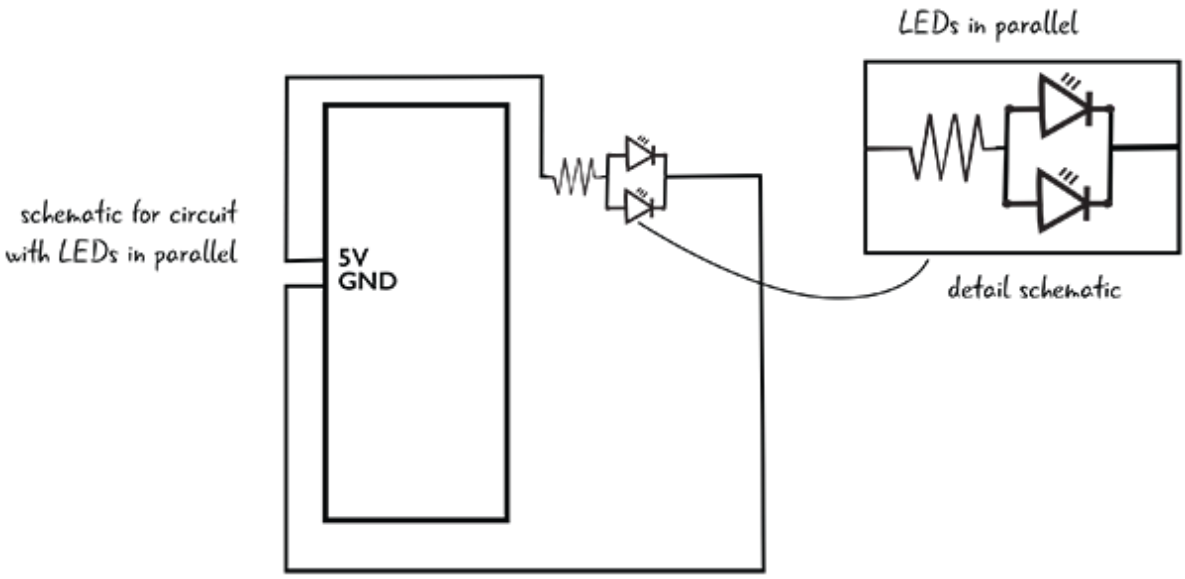


FIGURE 5-42: Schematic for circuit with LEDs in parallel

Add a Second LED to the Circuit

To create this circuit, add a second LED to the breadboard so that the anodes of both LEDs are in one row of connected tie points and the cathodes are in a different single row of connected tie points. Both of the anodes are now connected to one end of the resistor, and both of the cathodes are connected by a jumper to ground, as shown in Figure 5-43. Remember to disconnect your computer before you make any changes to the circuit.

You've added the second LED, so you're almost ready to check the voltage in this circuit.

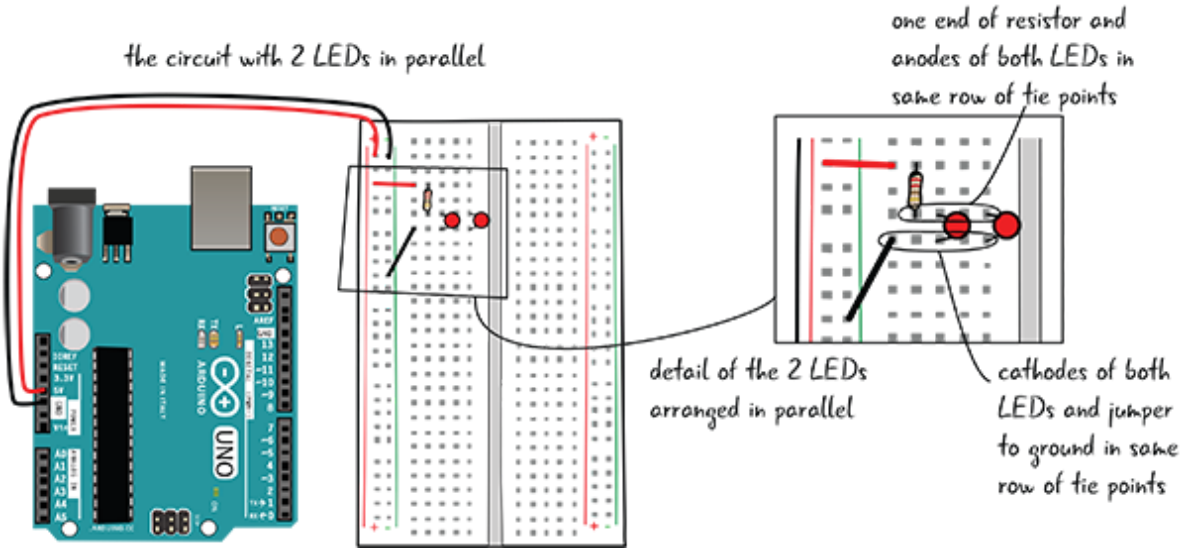


FIGURE 5-43: Adding the second LED in parallel

MEASURING VOLTAGE ACROSS LEDS IN PARALLEL

When you have the LED placed correctly in the breadboard, attach your computer again to the Arduino. Next, set the dial on your multimeter to measure 20 volts. Then place the red probe on the anode of one of the LEDs, and the black probe on the cathode of the same LED, as shown in Figures 5-44 and 5-45.

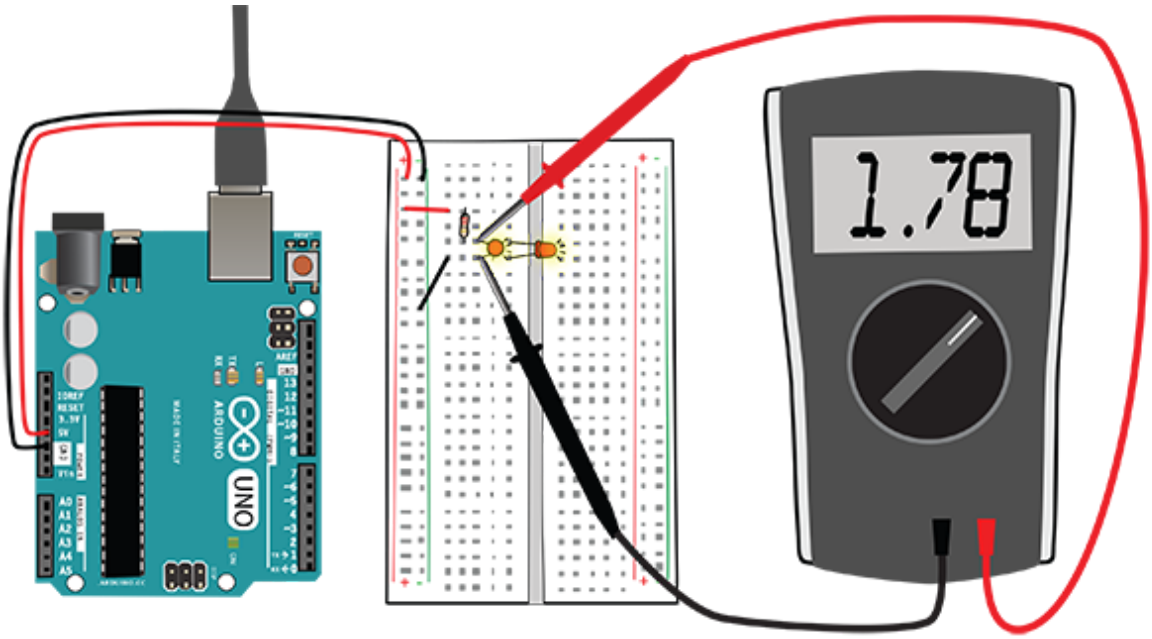


FIGURE 5-44: Measuring voltage of a component in parallel

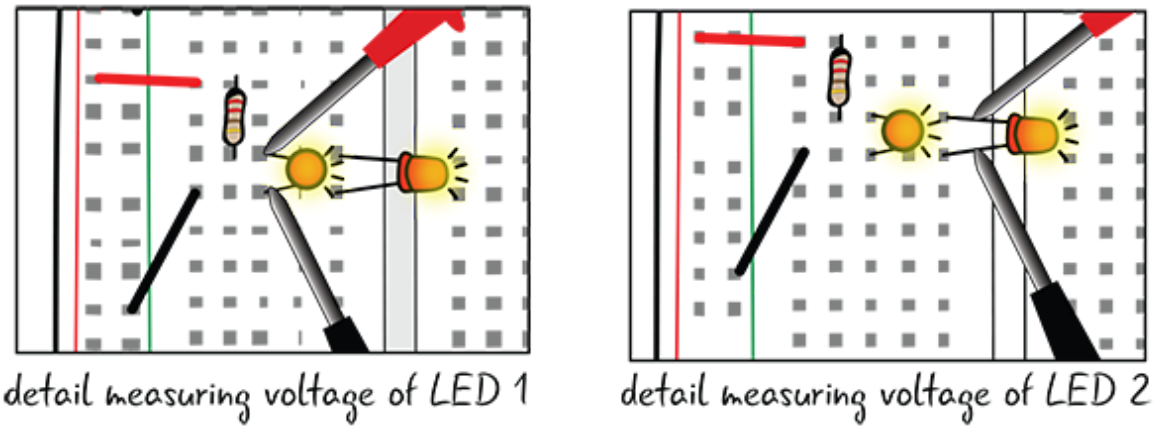


FIGURE 5-45: Measuring the voltage of LED 1 and 2

The display on your meter should read about 1.78 volts for red LEDs (if it is not exactly the same, that's because the LEDs you're using are rated differently than the ones you used to build the circuit). After you've tested the voltage across one of the LEDs, check the other, as shown in Figure 5-45. The value should be the *exact same voltage drop* for both LEDs if you used identical LEDs. You don't

need to measure the voltage across the resistor, because it will be the same value as you had with your basic circuit.

> **Note**
>
> In parallel, both LEDs receive the same voltage.

THE MULTIMETER IN PARALLEL

You may have noticed that the LEDs share common electrical contact points, and so does the multimeter. When you're using the multimeter to measure voltage, the multimeter is in a parallel arrangement with the component whose voltage you are measuring (Figure 5-46).

> **Note**
>
> Measuring voltage in a circuit places the multimeter in parallel with the component being measured.

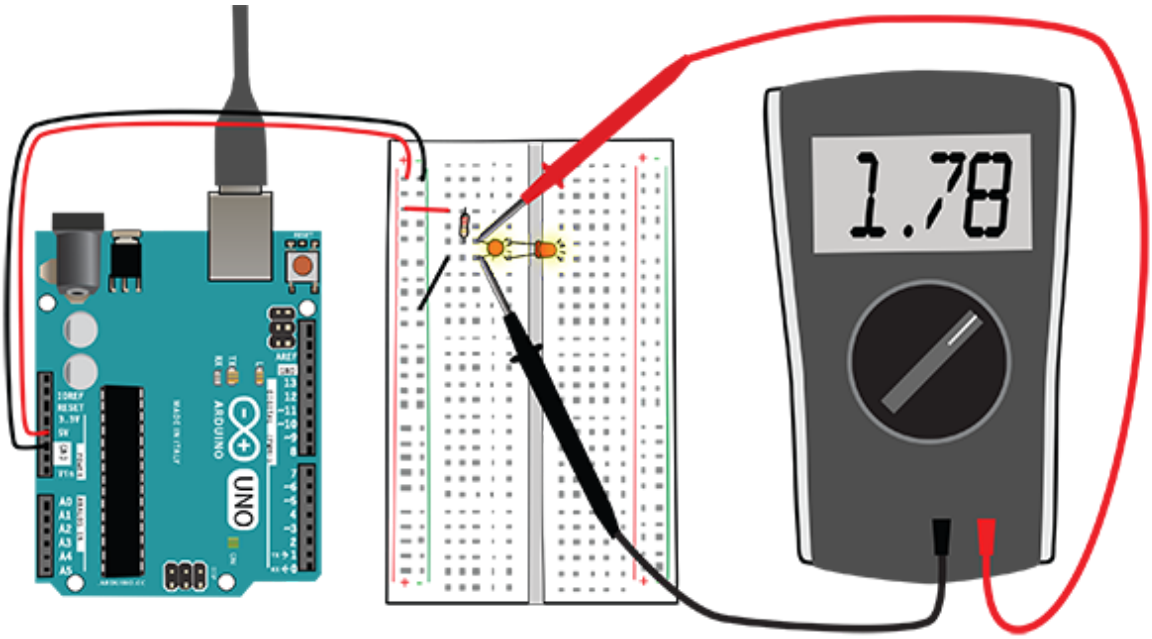


FIGURE 5-46: The multimeter is in parallel with the LEDs.

COMPONENTS IN PARALLEL: WHAT EFFECT DOES THAT HAVE ON THE VOLTAGE?

You know that components in parallel share the same electrical contact points. Electricity will take all possible paths from the beginning of a circuit to its end. As you saw with our voltage measurement, the same voltage will pass through all the components in parallel (Figure 5-47).

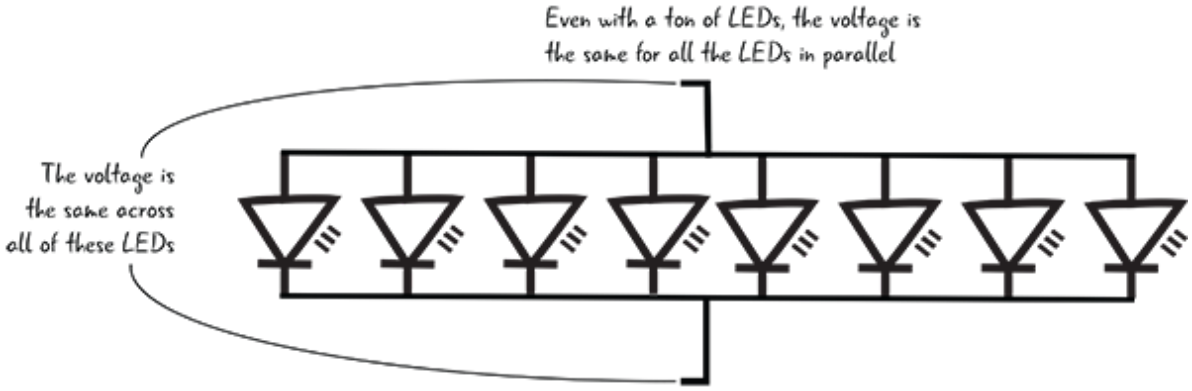


FIGURE 5-47: Many LEDs in parallel

If you want several LEDs to glow the same brightness, you can place the LEDs in parallel and know that they will all receive the same voltage, unchanged by the number of LEDs in place. You aren't able to light more than a few LEDs in parallel from your Arduino, however, since you're limited by the amount of current provided.

> **Note**
>
> Equal voltage will pass through all components that are in parallel.

BUILDING A TWO-LED SERIES CIRCUIT

Now we're going to adjust our circuit, placing the LEDs so they are in an arrangement called *series*. Components that are in series are placed one right after each other. It is easy to see this by looking at Figure 5-48, where two LEDs are shown one right after the other; in fact, the resistor is also in series in this arrangement. Most circuits will be a combination of components arranged in series and in parallel.

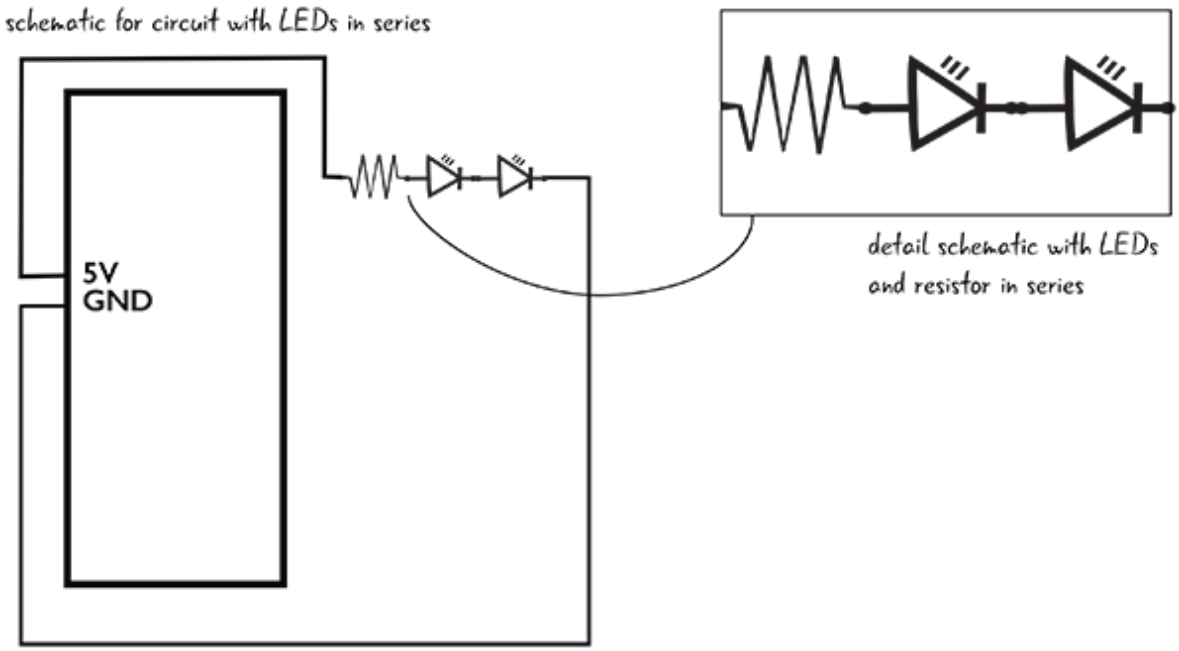


FIGURE 5-48: Schematic for the circuit with LEDs in series

Start by unplugging your Arduino from your computer. To place the second LED in series with the first, place the anode (long leg) of the second LED in the same row of tie points as the cathode (short leg) of the first LED. Place the cathode of the second LED in a separate row of tie points. You will have to move the jumper that goes to ground so it is in the same row of tie points as the cathode of the second LED, as shown in Figure 5-49. (Remember, the circuit has to be a complete loop in order to work.)

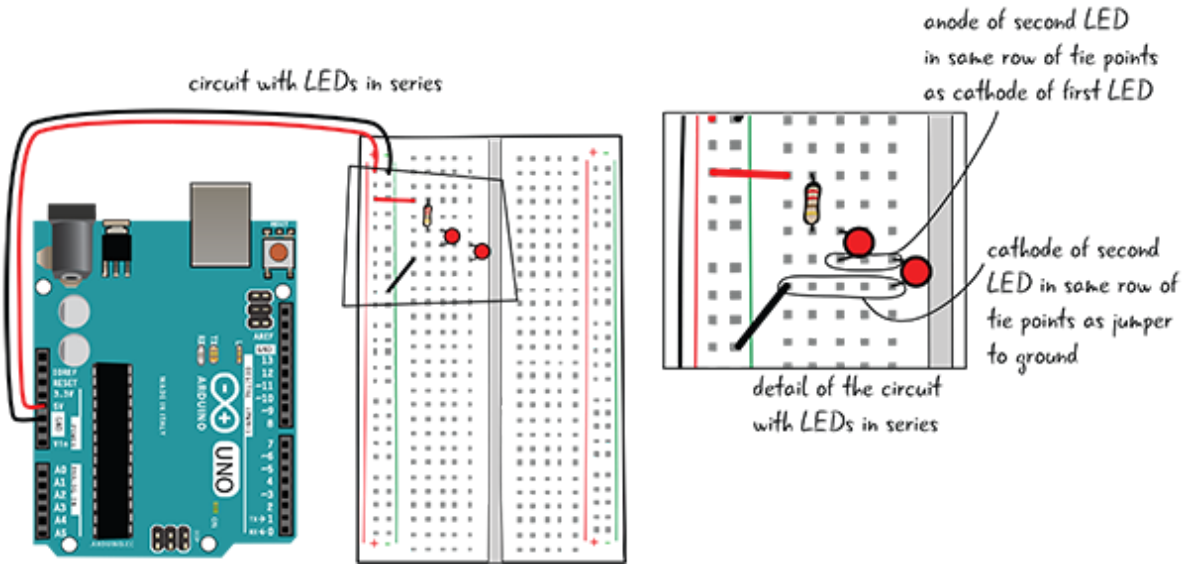


FIGURE 5-49: Placing the second LED in series

When you have adjusted your circuit, you will be ready to meter it.

METERING THE VOLTAGE OF COMPONENTS IN SERIES

Metering components that are in series for voltage is much the same as metering components that are arranged in parallel. Plug your Arduino into your computer and the LEDs should both light up. With the dial again on 20V, place the red probe on the anode of one of the LEDs and the black probe on the cathode of the same LED, as shown in Figure 5-50. The voltage should read something like 1.77 volts. Next, measure the voltage across the other LED; your result should be similar to what you got for the first LED. Finally, measure the voltage across the resistor (for us, the value was 1.38 volts). Figure 5-51 shows a detail of metering the components in the circuit.

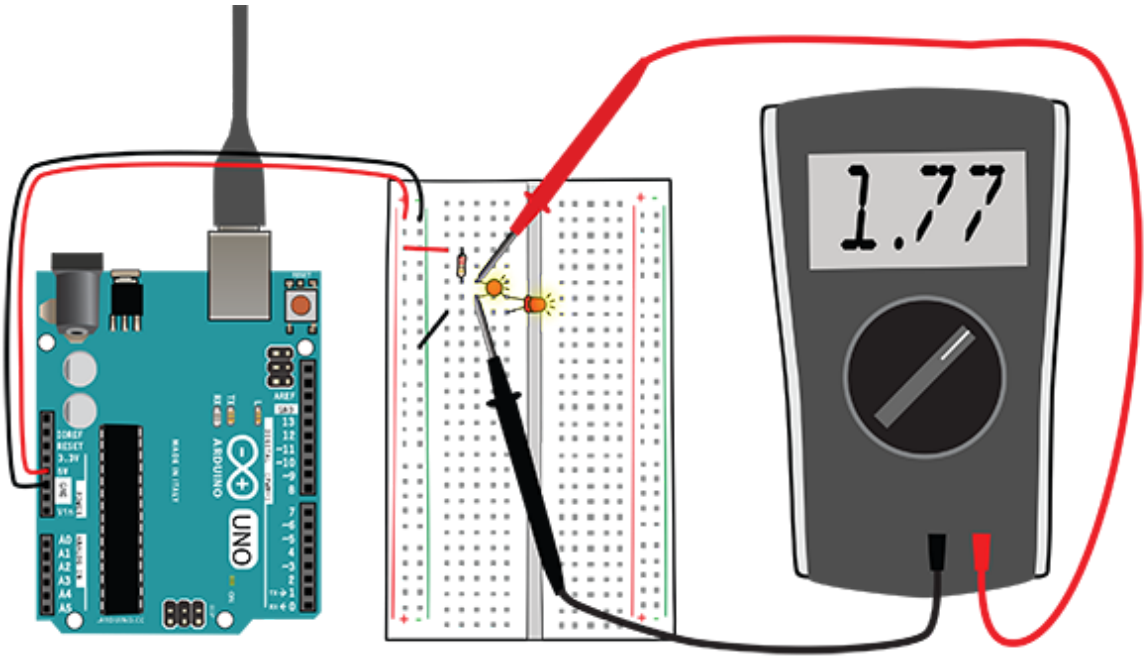


FIGURE 5-50: Measuring the voltage of the components in series

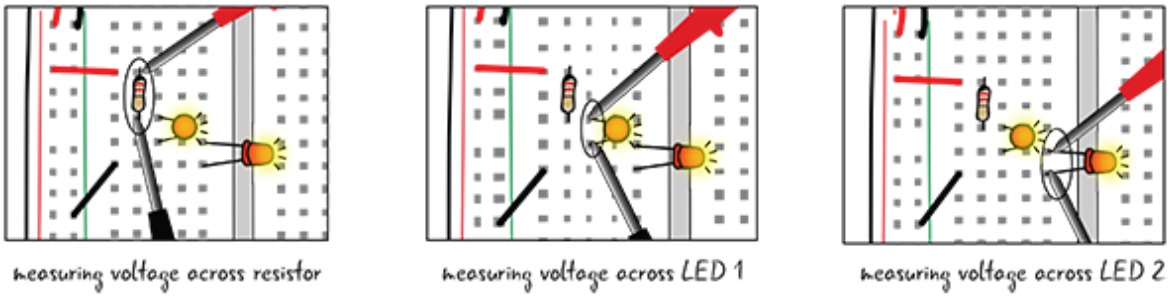


FIGURE 5-51: Details of metering the circuit

QUESTIONS?

Q: Why are the values for voltage in both the series circuit and the parallel circuit so similar?

A: In this case, we are not seeing as big a difference in voltage for series and parallel as we might in a different circuit. We felt that it was important to show you the differences in the way these components are arranged and how electricity flows in these circuits, and help you become more familiar with using the meter.

COMPONENTS IN SERIES: WHAT EFFECT DOES THAT HAVE ON THE VOLTAGE?

In your circuit, the electricity must pass through the resistor before it gets to the first LED. As you saw in our multimeter measurements, voltage is consumed as it passes through each component. Although the voltage across each LED is about the same as for your one LED in your basic circuit, the value of the voltage across the resistor drops. The resistor consumes less voltage in this series example because two LEDs in the circuit are consuming voltage. Note that the value of the resistor does not change, but since each LED now requires its own voltage, the resistor consumes a smaller portion of the total voltage. Figure 5-52 represents the voltage drop in the circuit. The values of voltage are each adjusted according to Ohm's law and can be measured with a multimeter.

You'll often have to wire resistors in series with other components, like LEDs, in order to drop the value of the voltage that enters your component.

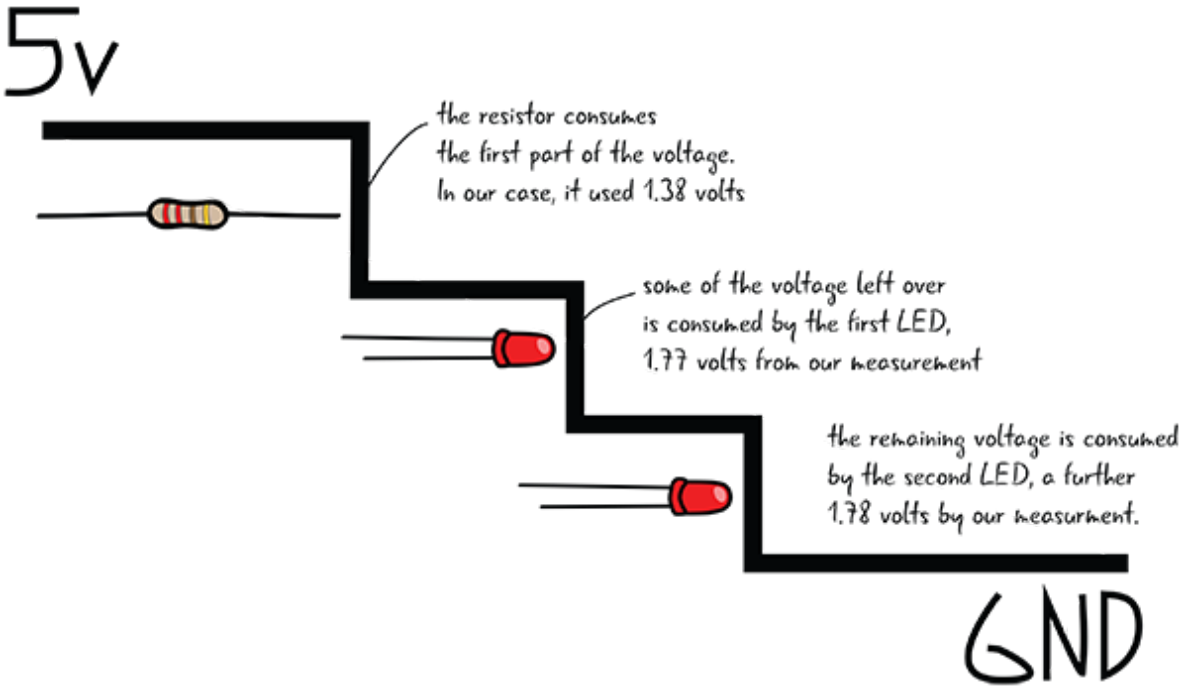


FIGURE 5-52: Visualizing voltage drop across the circuit with LEDs in series

It's less likely that you'll put multiple LEDs in series since each additional LED makes all your lights dimmer. Old strings of holiday lights, such as those in Figure 5-53, are a real-world example of lights designed to be wired in series. Being wired in series is the reason that if one bulb burns out the whole string of lights turns off. More recent string lights have been redesigned to avoid this problem.

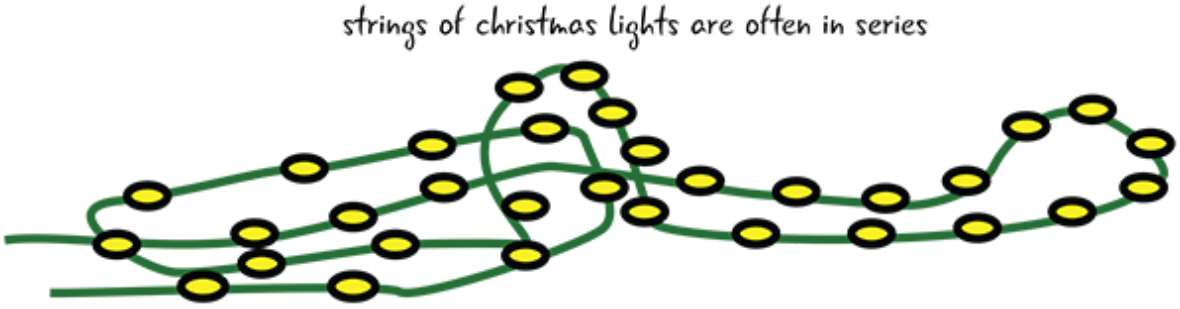


FIGURE 5-53: Christmas lights, arranged in series

THE MULTIMETER IN SERIES

Remember how you pulled out the anode of the LED when you were measuring current in your basic circuit? Then you inserted the multimeter right into the circuit, touching one end of the resistor and the anode of the LED to complete your circuit. In that arrangement, the multimeter was in series with the resistor and the LED. The multimeter has to be in series to measure the current, because then it does not alter the value of the current (Figure 5-54).

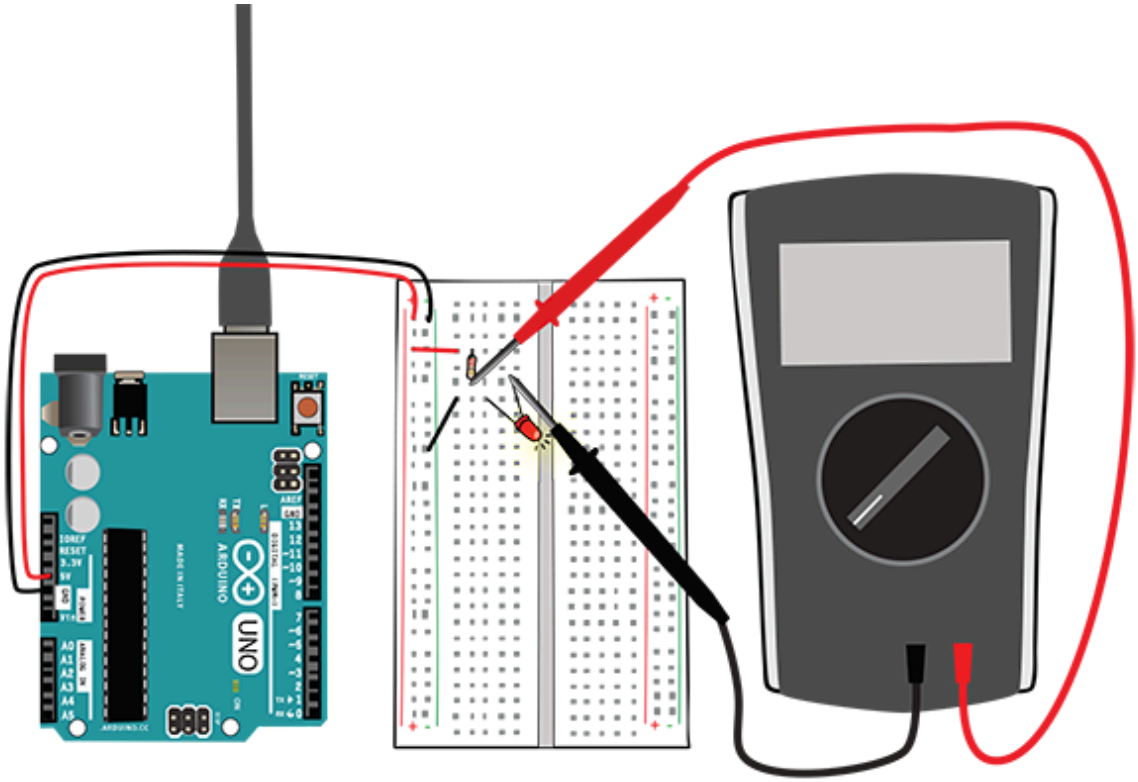


FIGURE 5-54: The multimeter is in series with the other components when measuring current.

Table 5-3 shows the effects of electrical properties on components in both series and parallel.

TABLE 5-3: Effects of electrical properties on components in series and parallel

EFFECT ON ELECTRICAL PROPERTIES	COMPONENTS IN SERIES	COMPONENTS IN PARALLEL
Effect on voltage	Each component consumes part of the voltage.	Equal voltage will cross all parallel components.
Effect on current	Equal current crosses all series components.	The current gets split based on the resistance value of each component.
Effect on resistance	Total resistance equals all values of resistance added together.	Resistance is reduced when components are in parallel.

SUMMARY

You've learned about voltage, current, and resistance and how they interact, through Ohm's law, and you know how to measure those properties with your multimeter. You've also learned about setting up components in series and in parallel. In the next chapter you'll return to Arduino projects and get additional programming practice.

6

SWITCHES, LEDS, AND MORE

In this chapter, you're going to learn how to make your projects interactive, first by adding a button to turn an LED on and off. Then, you'll attach a speaker to your Arduino and control both sound and light with your sketch. Finally, you'll add two more buttons, in the process of building a keyboard instrument on which you can play simple tunes. Throughout these projects, you'll learn more about programming the Arduino. To complete the projects in this chapter, you need to have the Arduino IDE installed, know how to connect a breadboard to your Arduino, and be familiar with writing a sketch and uploading it to your Arduino.

INTERACTIVITY!

In Chapter 4, "Programming the Arduino," you saw how to connect the Arduino to a breadboard to build a circuit that lit up an LED in an SOS pattern. The LED turned on and off, over and over and over

again, always in the same pattern. Wouldn't it be great if you could build a circuit that would respond to the user's action?

You'll do just that in this chapter by building a mini-keyboard instrument with three buttons, a speaker, and an LED. The speaker will play a different tone, depending on which button you push, so you can play a tune. And the LED will turn on whenever you push any of the buttons. We'll start with a circuit with an LED and a button and build on that. Figure 6-1 is a preview of what the finished circuit will look like.

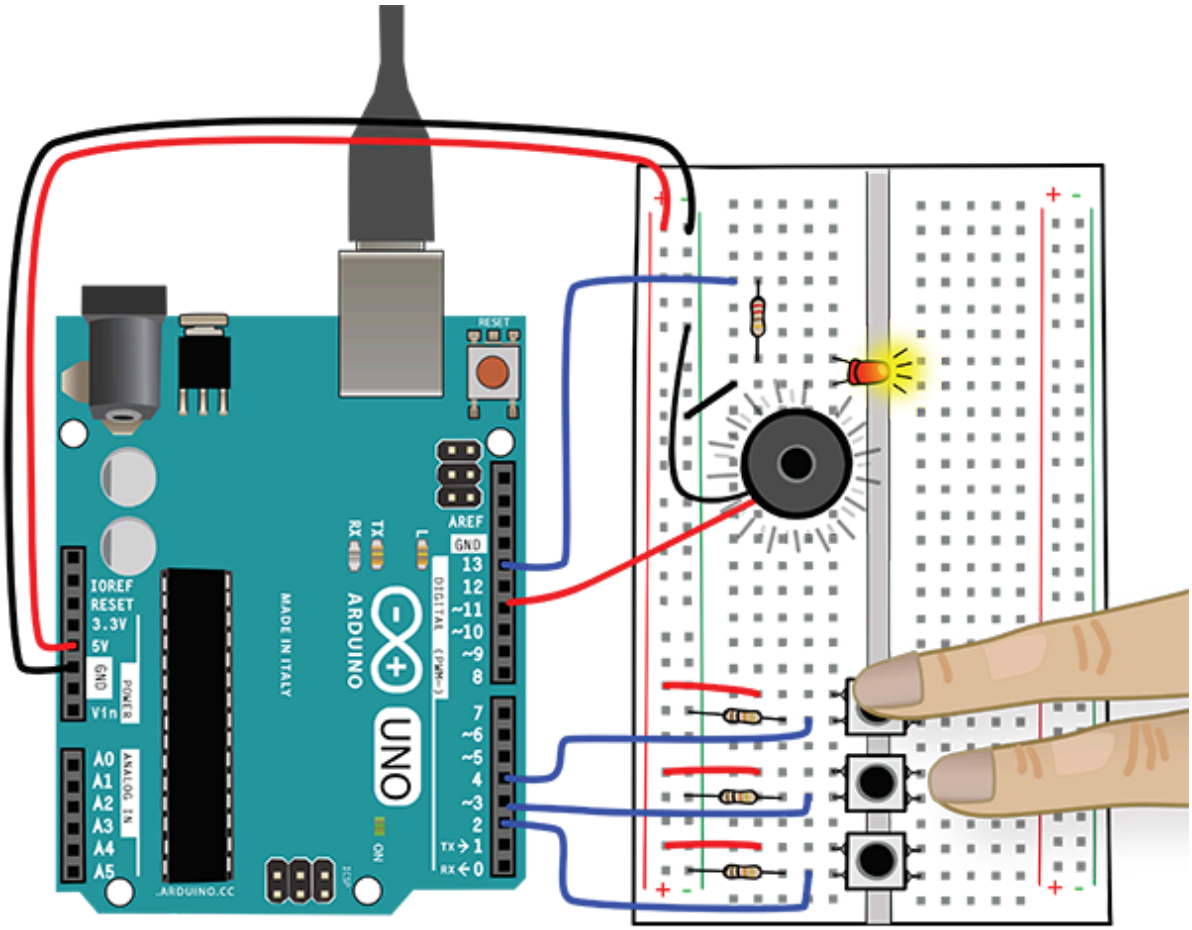
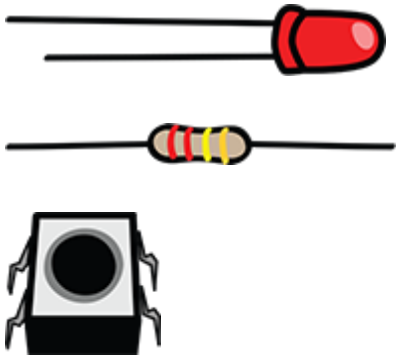


FIGURE 6-1: Three-tone button keyboard

Just as in Chapter 4, our project will consist of an Arduino and breadboard circuit with code written in the Arduino IDE. We'll go over

building the circuit and all of the code in the sketch step by step. In this chapter, you will also learn a bit more about reading schematics.



This project uses digital inputs and outputs. You used a digital output, an LED, in the last chapter. Before we start to build, let's look more closely at what we mean by digital input and output.

DIGITAL INPUTS AND OUTPUTS OVERVIEW

Think about your computer: how do you get information into it? You may use a mouse, and you probably use a keyboard. Keyboards and mice are both examples of *inputs* (Figure 6-2). You can attach many different types of inputs to your Arduino. In this chapter you will attach buttons to the Arduino as inputs.

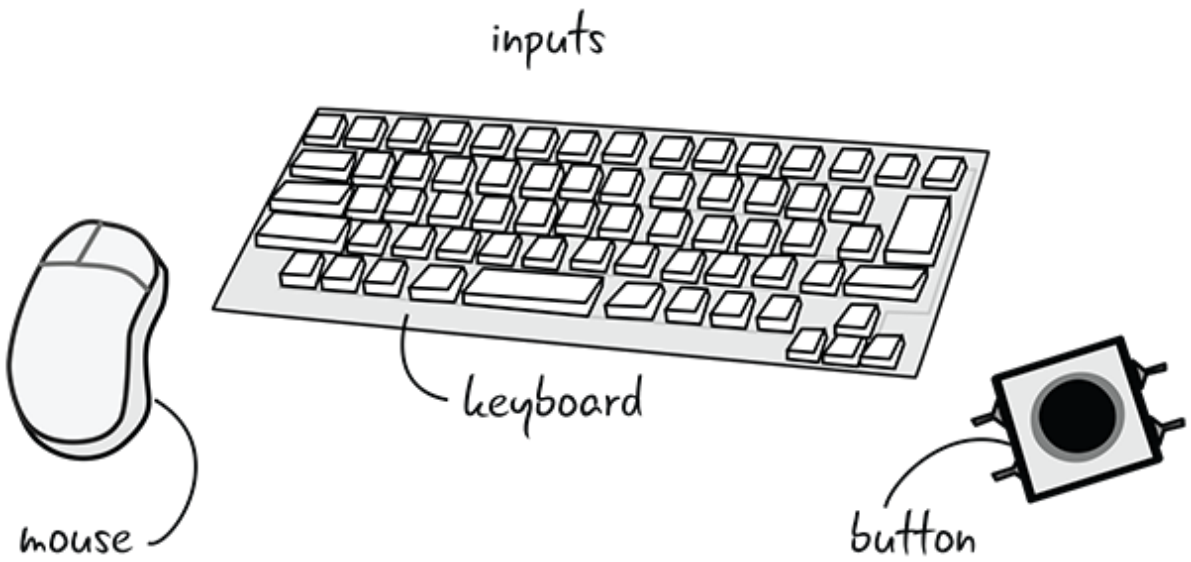


FIGURE 6-2: Common inputs

What do we mean by output? Again, think about your computer. You might have speakers attached to it, or a monitor or printer. Those are all examples of *output* devices (Figure 6-3). An Arduino can have many different types of outputs attached to it. In fact, you have already used an output device with the Arduino: the LED you connected to it in the last chapter.

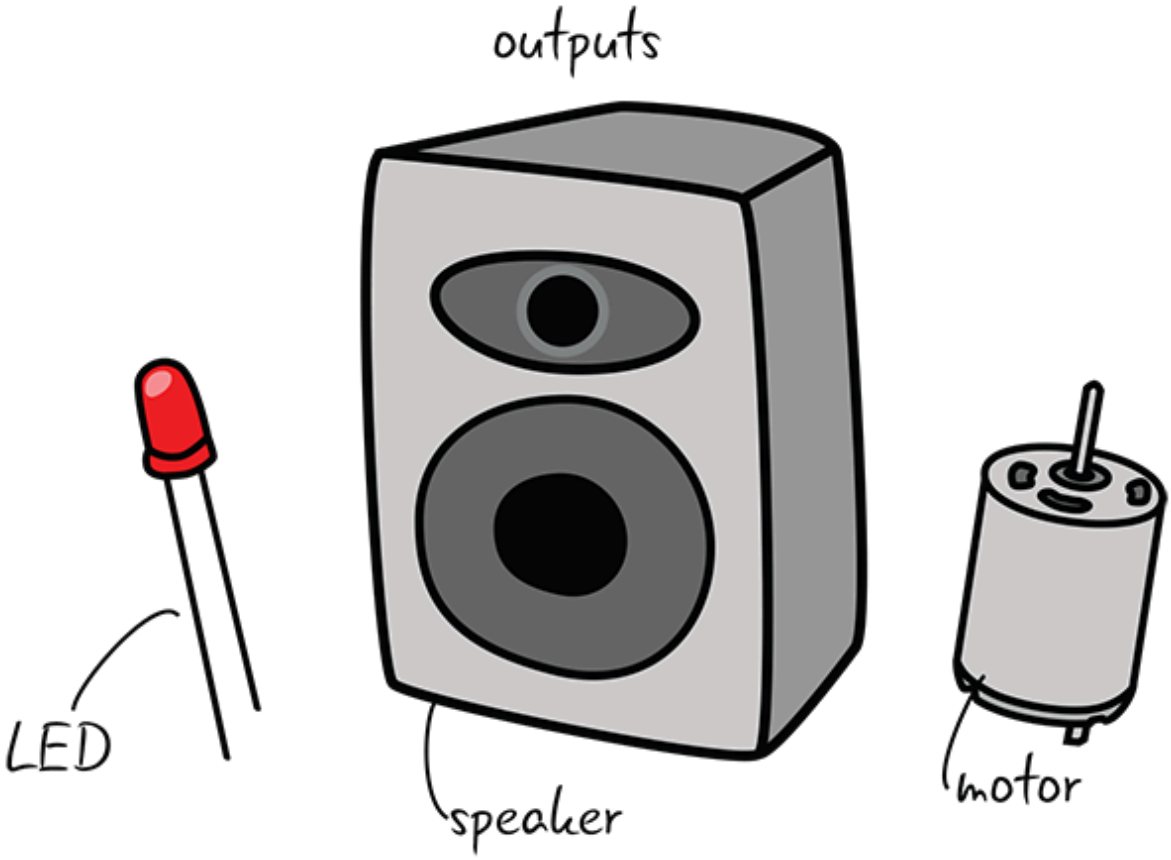


FIGURE 6-3: Common outputs

For now, think of digital inputs and outputs as components that have only two possible states: on or off. Inputs send messages to the computer. Outputs receive messages from the computer. We will explain this in more detail later in the chapter.

Before you start building your circuit, let's look at the schematics for a button or switches. Doing so will help you understand how digital input works.

SWITCHES

There are a million different ways to trigger electronic devices or turn something on. Switches and similar on/off devices activate televisions, music equipment, lights—even your kitchen appliances! How does a switch work?

All switches work on the same basic principle: it either "closes the circuit" and turns something on, or "opens the circuit" and turns something off. When the switch is closed, electricity can flow through; it cannot flow through when the switch is open. Figure 6-4 illustrates how this works.

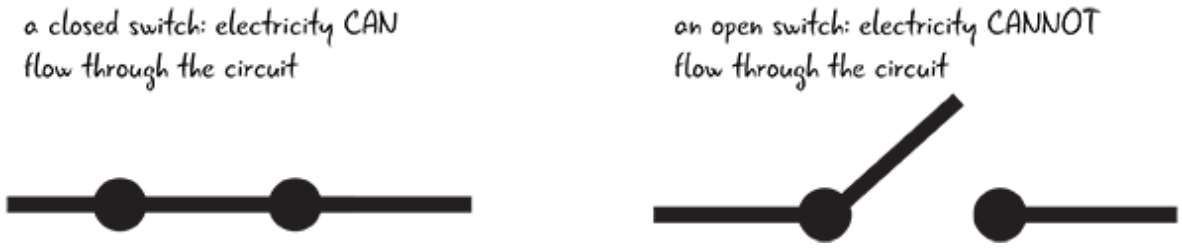


FIGURE 6-4: Switch diagrams

Like all digital inputs, as you saw earlier, switches have only two possible states: on and off. In the Arduino IDE, on and off (respectively) are equivalent to `HIGH` and `LOW`. (Remember how, in the SOS circuit in Chapter 4, the light turned on when you set it to `HIGH` and off when you set it to `LOW`.) Each key on a keyboard is actually a switch, set in the off position until pressed down, when it goes to on.

Buttons are one type of switch. For our circuit, we will use a momentary pushbutton switch, which closes and completes the circuit when you press it. As soon as you let go, the switch opens again and the circuit is no longer complete.

DIGITAL INPUT: ADD A BUTTON

Let's get our parts together to start building a circuit with a button.

You'll need the following:

1 LED

1 220-ohm resistor (red, red, brown, gold)

1 10 kΩ resistor (brown, black, orange, gold)

1 momentary pushbutton switch

Jumper wires

Breadboard

Arduino Uno

USB A-B cable

Computer with Arduino IDE

Figure 6-5 shows a preview of what the circuit will look like when it's built, as well as the schematic. Since the schematic is a bit different from those we have seen before (it includes some new symbols), we will take a closer look at it.

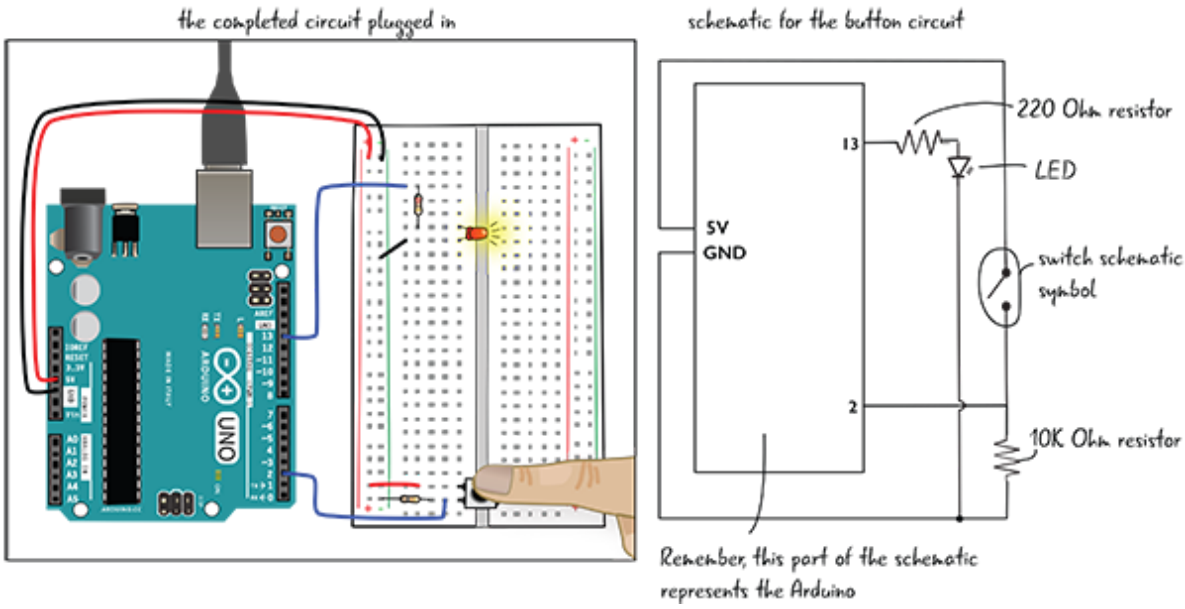


FIGURE 6-5: LED button circuit

UNDERSTANDING MORE COMPLEX SCHEMATICS

The schematic for this circuit follows a couple of conventions that you haven't seen before.

At the bottom of this schematic is a small circle indicating that the cathode of the LED (remember: the cathode is the LED's short lead or negative side) on Pin 13 is connected to the same ground as the

resistor that is attached to one end of the switch. A filled circle is often used in schematics to indicate connection points.

As schematics become more complex, you sometimes have to run lines that are *not* connected over each other. To indicate that these lines are not connected, we'll draw a little loop like the one on the right side of Figure 6-6.

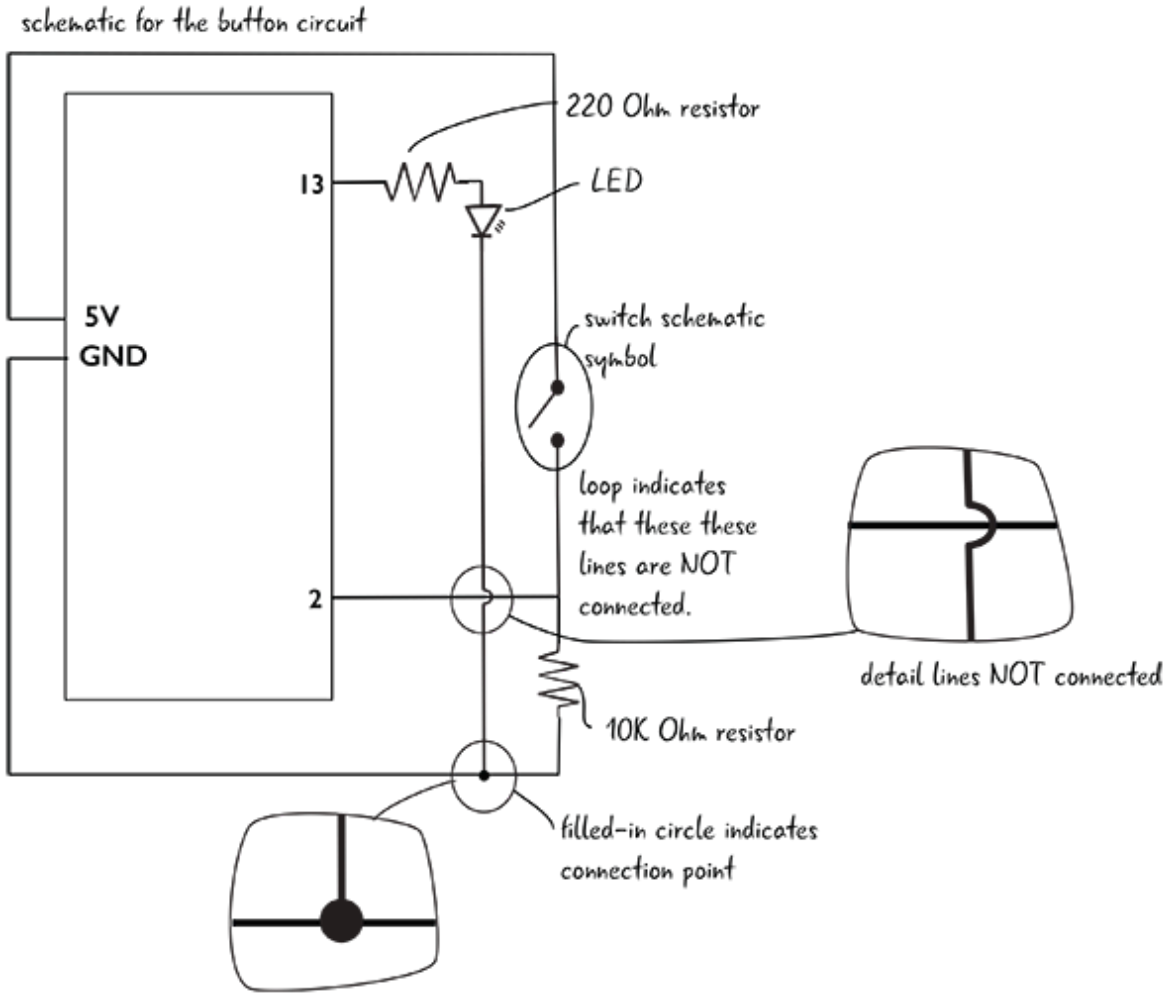


FIGURE 6-6: Schematic diagram

BUILDING THE BUTTON CIRCUIT

Before you add the button to the circuit, you have to rebuild the circuit you used with the LEA4_Blink sketch in Chapter 4, shown

again in Figure 6-7. Here's a quick overview of how to do that. Follow along and check each step as you go:

Attach the power and ground from the Arduino to the power and ground buses on the breadboard.

Connect a jumper from Pin 13 on the Arduino to a row of tie points on the breadboard.

Connect a 220-ohm resistor to the same row of tie points as Pin 13.

Connect the anode (long leg) of the LED to the other end of resistor, and jump the cathode (short leg) of the LED to the ground bus.

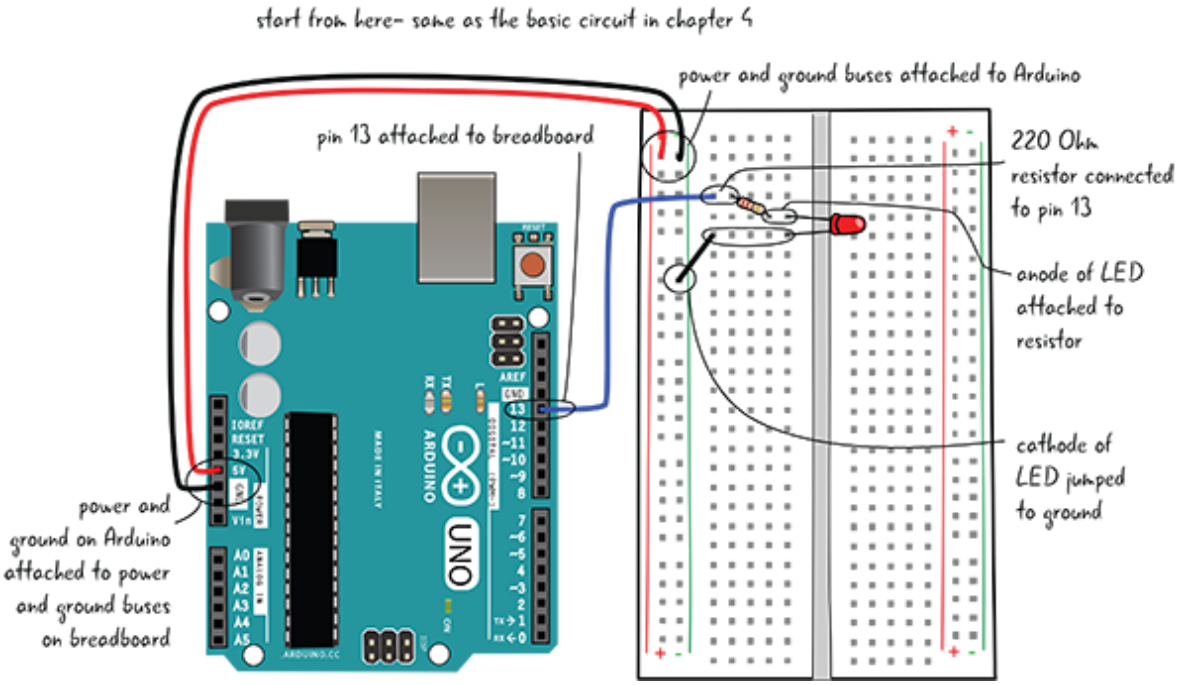


FIGURE 6-7: Reviewing the basic circuit from Chapter 4

Now that you have the basic circuit assembled, you'll add a button. You've looked at the schematic for a button, but before you put the button in the breadboard, let's look at how the button is constructed.

ADDING THE BUTTON

The button you're going to add is a pushbutton switch (a.k.a. a momentary switch). While you are pressing the button, the LED will turn on, and as soon as you lift your finger, the LED will turn off (Figure 6-8). This type of button gets its name from just being toggled for the moment.

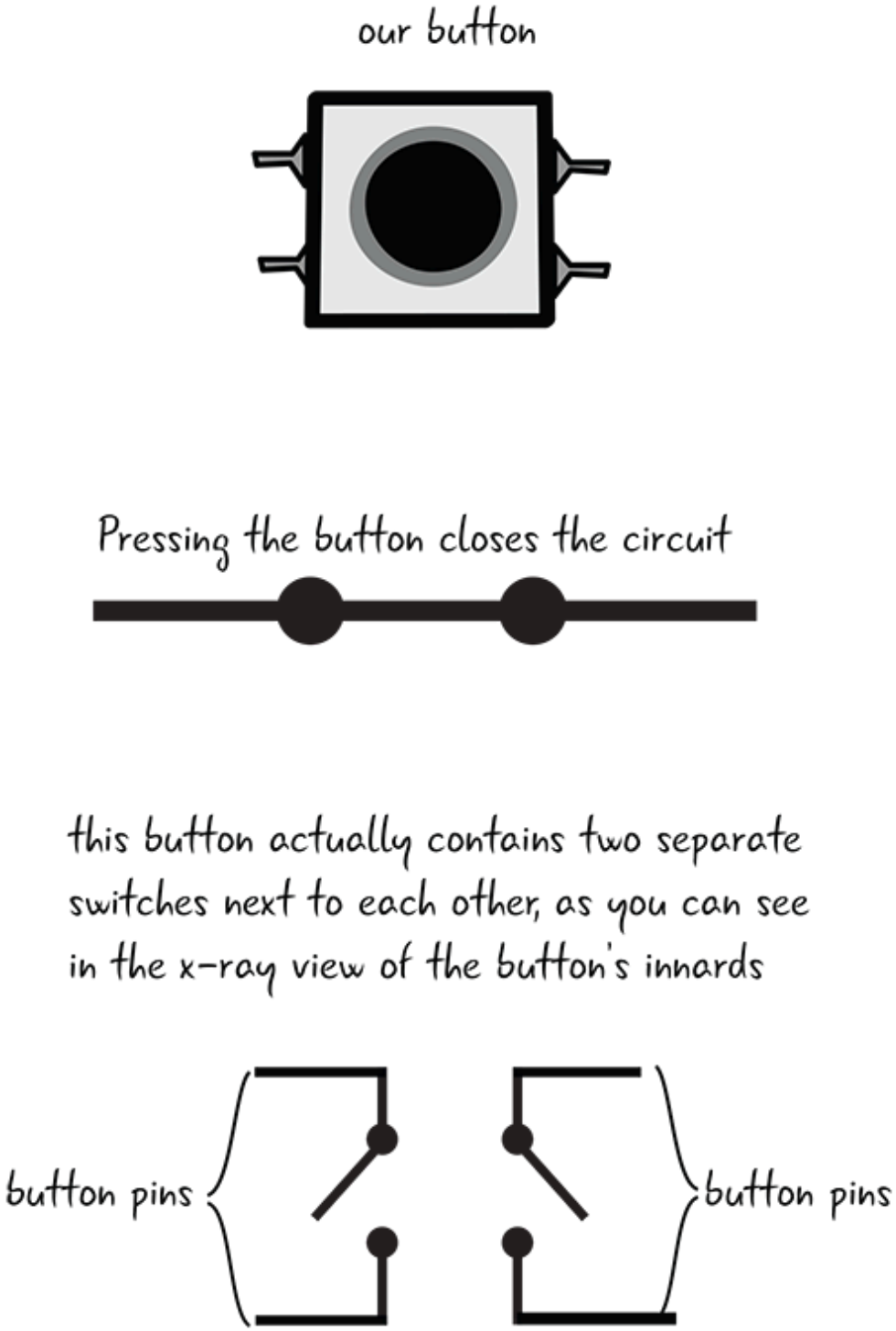


FIGURE 6-8: Button diagrams

When the button is pressed, the circuit is closed, and electricity can flow through it, like the diagram you saw earlier in this chapter.

This button actually contains two separate switches (that's why it has four pins sticking out of it). Both close when you press the button down. Your circuit will use only one side of the button.

Remember the trench that runs down the middle of the breadboard? (You learned about it in Chapter 3, "Meet the Circuit.") Your button is going to be placed across the trench, with two pins inserted into the row of tie points on each side. The button will only fit across the trench in one orientation. Placing your button across the trench ensures that it is oriented correctly, with each pin connected to a separate and discrete row of tie points (Figure 6-9). As long as the button fits across the trench, the button direction will not matter.

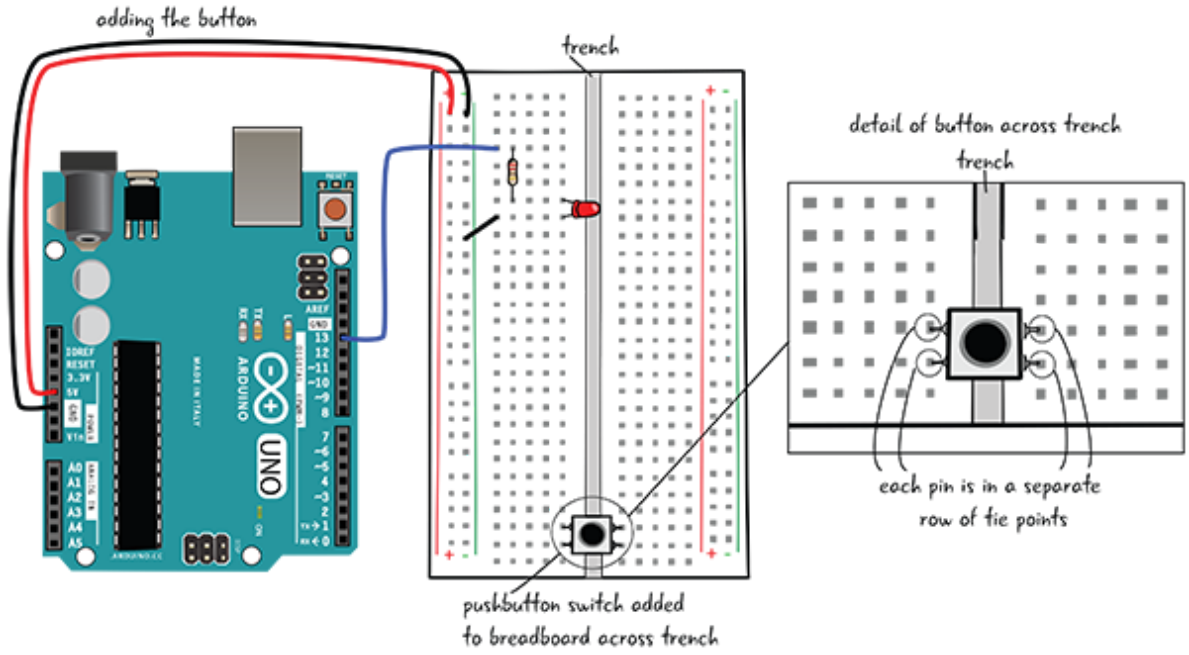


FIGURE 6-9: Adding the button to your breadboard

> **Tip**
>
> You are placing the button near the end of the board to make it easier to add all your components.

CONNECTING THE BUTTON TO POWER, A RESISTOR, AND GROUND

Let's continue wiring the button. Add a red jumper that connects the power bus (the one with the "+" sign) on the breadboard to the top-left side of the button (Figure 6-10).

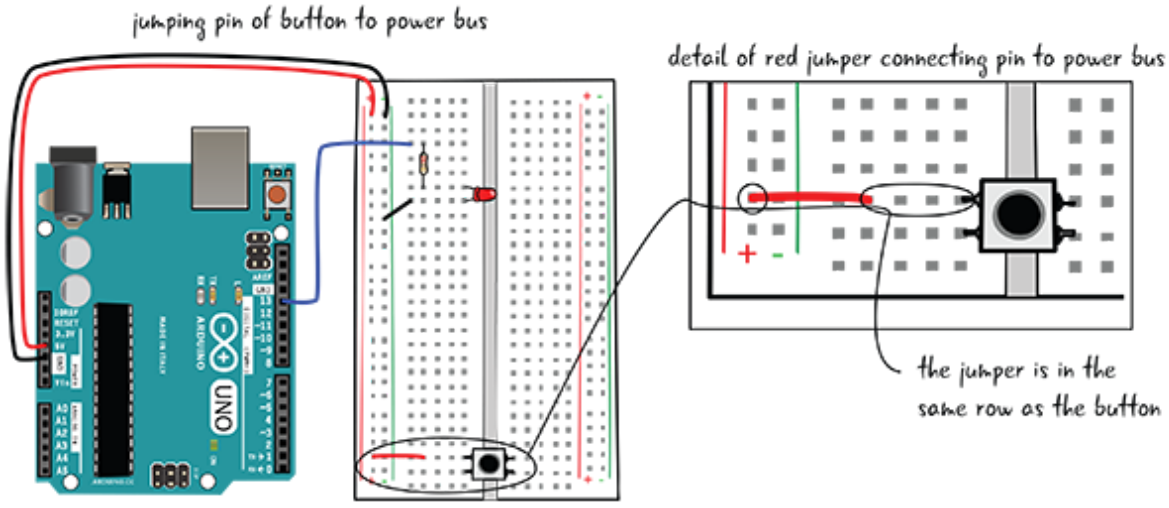


FIGURE 6-10: Adding the first jumper wire

Next, attach a 10 kΩ resistor (colored brown, black, orange, gold). Attach one of the resistor's leads to the button, and the other lead to the ground bus (the one with the "–" sign), as shown in Figure 6-11.

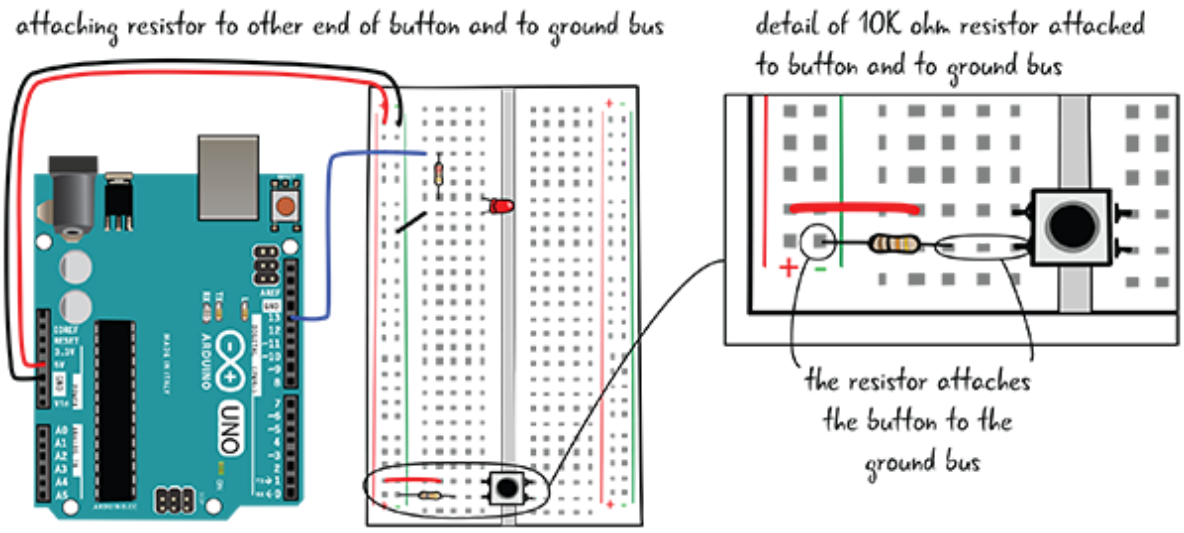


FIGURE 6-11: Adding the resistor for the button

ATTACH THE BUTTON TO AN ARDUINO PIN AND UPLOAD A SKETCH

Finally, you'll attach a jumper to Pin 2 on the Arduino. This jumper will be attached to the 10 kΩ resistor that is attached to the ground bus. As you can see in Figure 6-12, the resistor, the jumper to the pin, and one of the pins of the button should all be in the same row of tie points. The jumper wire also needs to be in between the resistor and the button.

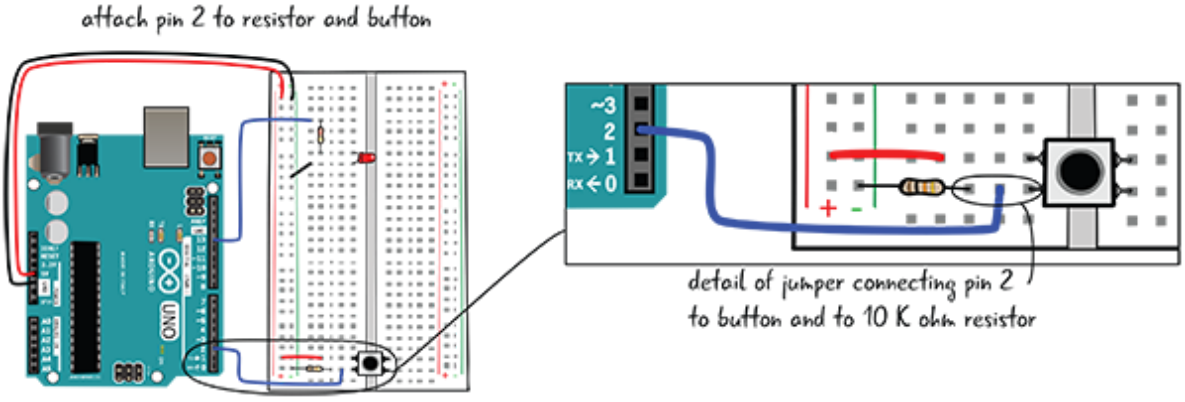


FIGURE 6-12: Adding the jumper to the digital pin

The button is all hooked up. Now that the Arduino is attached to the breadboard and the button is wired up, hook up the Arduino to your computer so you can upload a sketch that will control the behavior of the button and the LED.

OPEN, SAVE, VERIFY, AND UPLOAD

Attach your computer to the Arduino with the USB cable so you can upload the Button sketch. This is one of the example sketches that comes with the Arduino IDE.

Launch the Arduino IDE, and then open the Button sketch by choosing File > Examples > 0.2 Digital > Button.

Save the button sketch as LEA6_Button.

Click the Verify button first to make sure your code is okay.

Click the Upload button to upload your code to the Arduino. This is all shown in Figure 6-13.

FIGURE 6-13: Procedure for getting code onto the Arduino

TURN THE LED ON AND OFF

Now when you press the button, the LED will light up, as you can see in Figure 6-14.

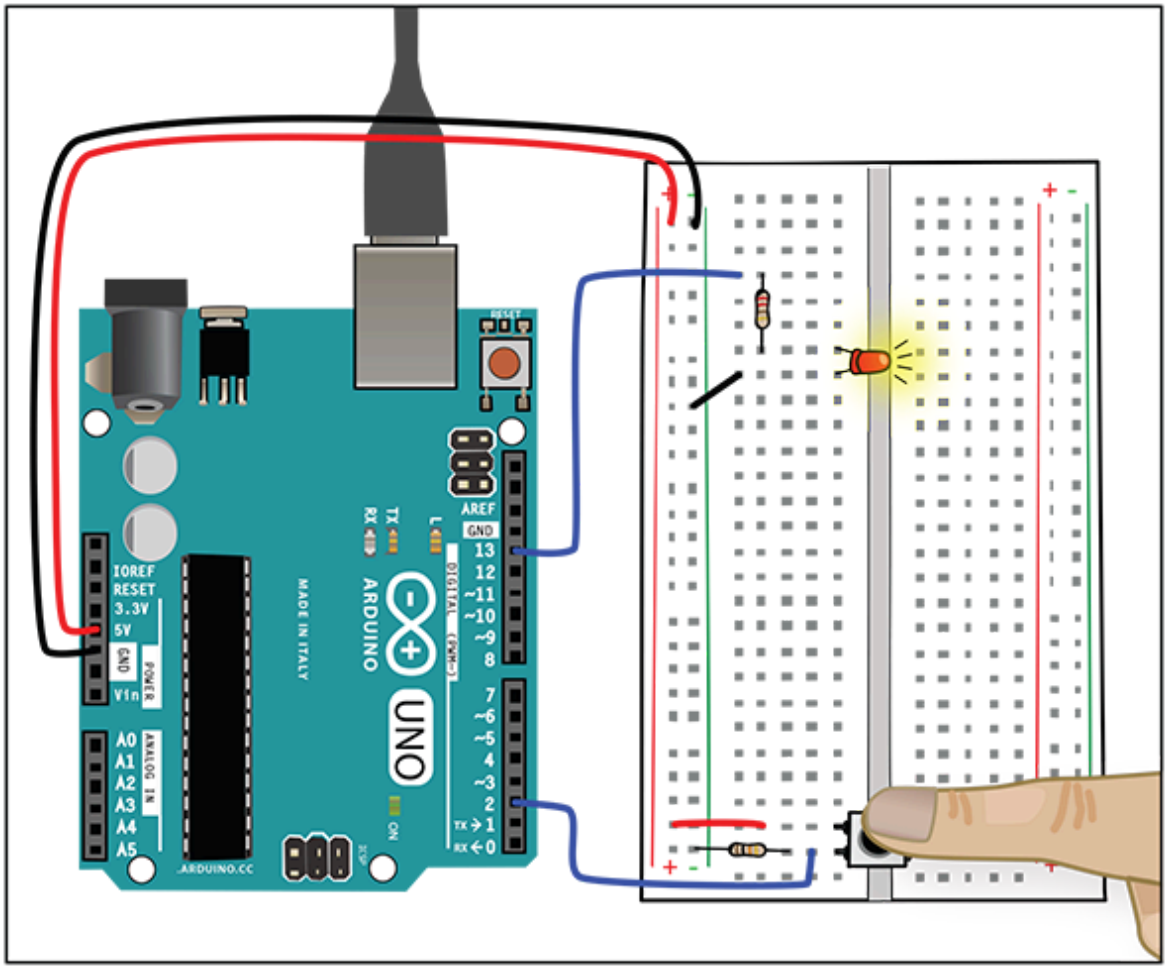


FIGURE 6-14: Press the button and the LED lights up.

QUESTIONS?

Q: Can I use other types of switches or buttons that I find?

A: Yes! All switches and buttons work on the principle of either closing the circuit (making a complete loop) or opening the circuit (breaking the loop).

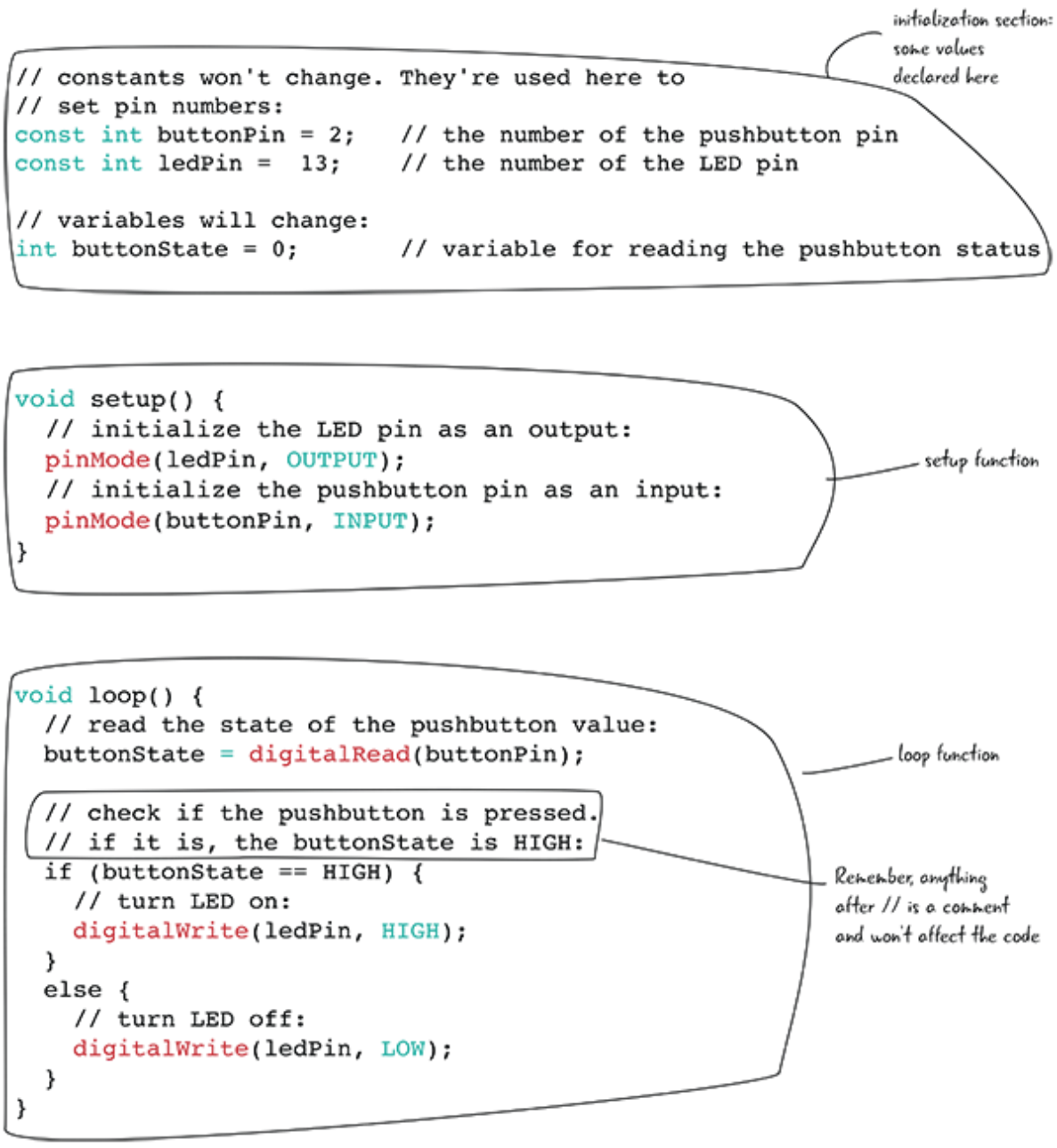
Q: Can I use a single button to trigger more than one output? For example, could I use one button to trigger a whole string of lights?

A: Although one button can be programmed to trigger many different things at the same time, most electronics have one button per function because that setup makes it easier for the user to understand exactly what is being triggered. If the same button triggers many functions, the user can be confused as to how the interaction works.

You've built the circuit and looked at the schematic. Now let's examine the code for LEA6_Button in detail.

LOOKING AT THE SKETCH: VARIABLES

Here's the code for the LEA6_Button sketch. We'll step through the details over the next few pages. We've removed the code's starting comments for the sake of brevity.



INITIALIZING OUR CODE AND VARIABLES

LEA6_Button is different from our LEA4_Blink sketch in that there is code that happens before the `setup()` function. This initial code is aptly called *initialization code*—code at the very top of a sketch where you declare values that you want to have access to throughout your

sketch. Let's take a look at this sketch's three lines of initialization code:

```
const int buttonPin = 2;    // the number of the pushbutton pin
const int ledPin =  13;     // the number of the LED pin

int buttonState = 0;        // variable for reading the pushbutton status
```

All three lines of codes look similar; they all have some words on the left and numbers on the right with an equals sign in the middle. But what do they do? To help you understand this section of code, we'll introduce a new programming concept: *variables*.

WHAT IS A VARIABLE?

In the simplest terms, a *variable* is a place to store a specific value and give it a useful name. Think of a variable as a container that holds a value. If you've taken algebra, you're familiar with variables. Remember equations where you were told things like "if x = 1"?

Variables can hold different types of values. For now we will look at variables that hold integers.

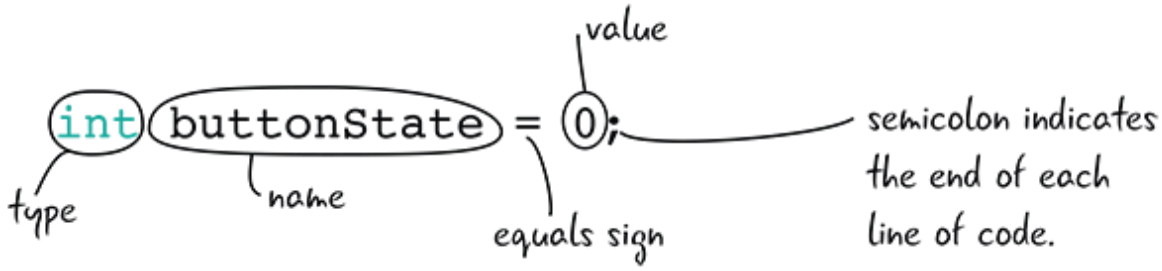
In the following code line, we are both *declaring* and *assigning* a variable. Declaring a variable means to give it a name, and assigning the variable gives it a value. You can declare variables without giving them a value, but you can't give a value to something you have not yet declared.

DECLARING VARIABLES

A line of code that defines a variable is called a *variable declaration*. Often Arduino code will also include a *variable assignment*. Variable

declarations and assignments are not structured the same in all programming languages—we are looking only at how variables are declared in the Arduino programming language.

In the Arduino language, all variable declarations and assignments have at least four different parts: the *type* of data the variable contains, the variable's *name*, an *equals sign*, and the actual *value* you wish to set for the variable. (Variables can have more than four parts; you'll learn about a fifth part momentarily.)



In this example, you are putting the value 0 into a variable with the name `buttonState` that has a type of `int`, which stands for integer. Let's look at all the parts of that declaration in detail. We'll start with the name, and then look at the value.

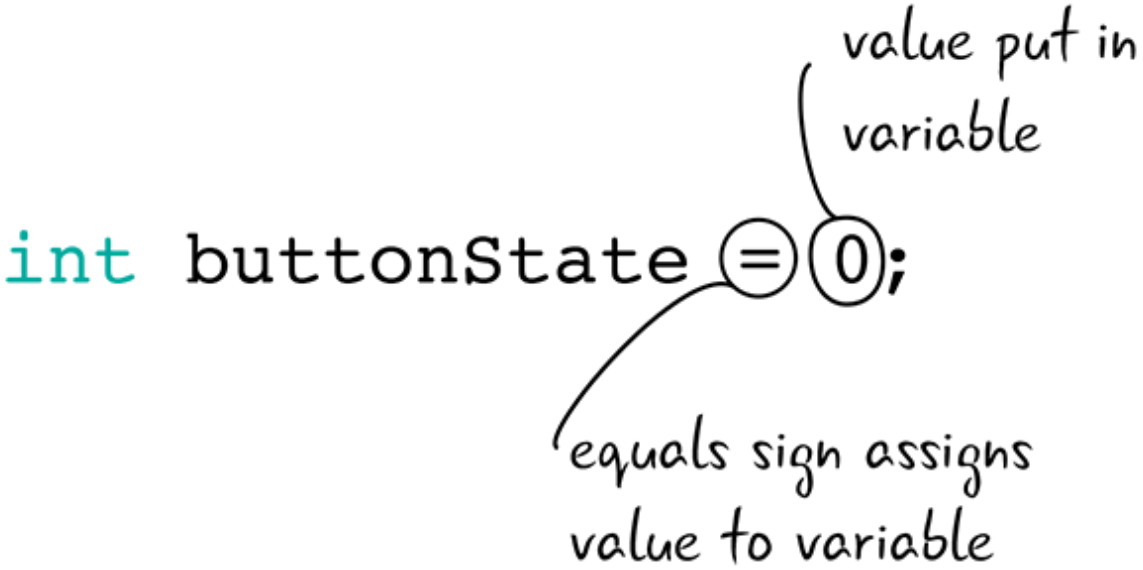
Variable Name

The `name` part of the declaration determines how you refer to your variable through the rest of your sketch. There are a few rules for selecting names: a variable name can't start with a number, can't have any spaces in it, and can't be a word that the Arduino language already uses for another purpose (for example, we can't name a variable "delay," since that term is already reserved by the Arduino language). There should be only one variable with a particular name for each sketch. It is considered best practice to name your variables something that indicates their purpose.

> **Note**
>
> Variable names can't start with a number, include spaces or symbols, or be a word used by the Arduino language.

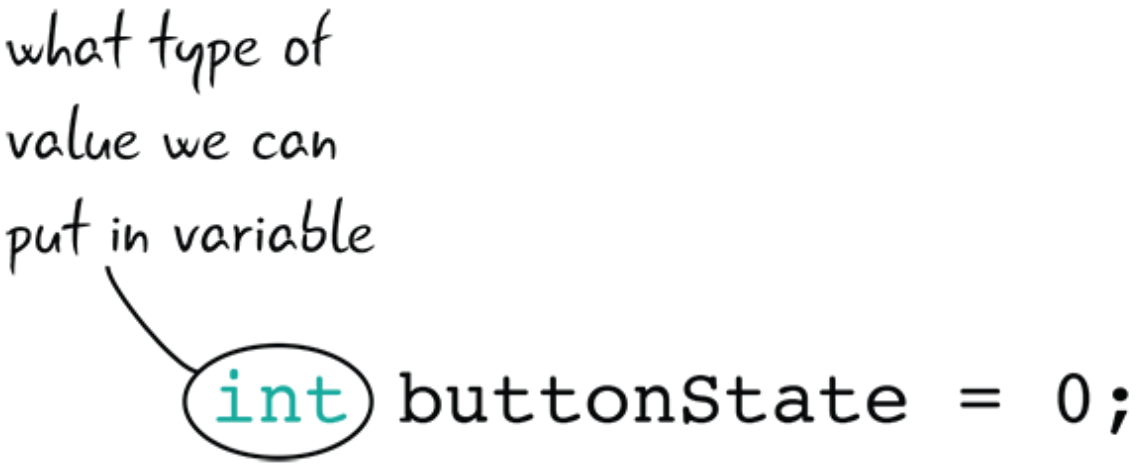
Variable Value

The `value` part of the declaration is what is stored in the variable. In this example you have an integer value of 0, which corresponds to the state, or voltage value, of our pin. Values are set using one `=` (equals sign), which says that anywhere you see this variable's name (`buttonState`), it means "Use the value 0" or "You are assigning the value 0 to this variable."



Variable Type

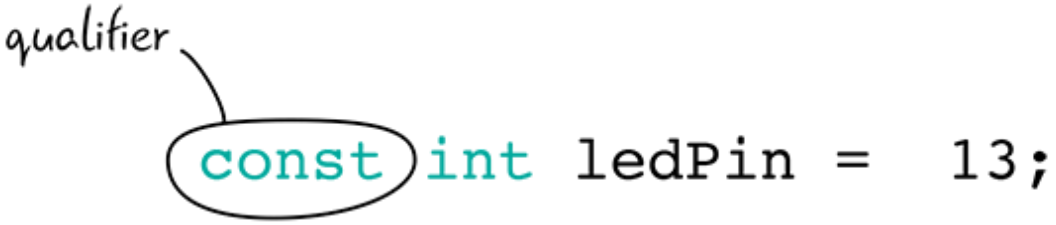
`type` sets what *type* of information you can save within your variable. In the declaration we're examining, `int` stands for *integer*, which means you can save only values that are whole numbers. Not all languages have typed variables, but in the Arduino language, you must declare the type of each variable in your sketch. Other types include float, string, character, and Boolean; you can learn more about variables types at arduino.cc/en/Reference/VariableDeclaration.



Now that you know about the four parts required in variable declaration, let's examine one optional part that's used in a couple of variable declarations in our sketch: the qualifier.

Variable Qualifiers

Some variables also have a qualifier, which determines whether you can change the value of the variable after you create it. The qualifier `const` sets your variable to have a permanent value when you run the sketch. In this context, `const` stands for *constant*. Here's an example from our sketch:



It may seem a little strange to think of a constant variable, but remember that setting a variable just means to *keep track of a value with a name*. When you plug the wires into the pins of your Arduino, the pin numbers are not going to change. Since the value of the variable isn't going to change, you add `const` to your declaration, which makes it clear that this is a constant variable.

Note

The qualifier is optional for variables; most variable declarations only have a type, a name, and a value.

As you can see from the following initialization code, our sketch contains two constant variables and one variable that can change:

```
const int buttonPin = 2;    // the number of the pushbutton pin
const int ledPin =  13;     // the number of the LED pin

int buttonState = 0;        // variable for reading the pushbutton status
```

QUESTIONS?

Q: What happens if I name my variable something that is not allowed?

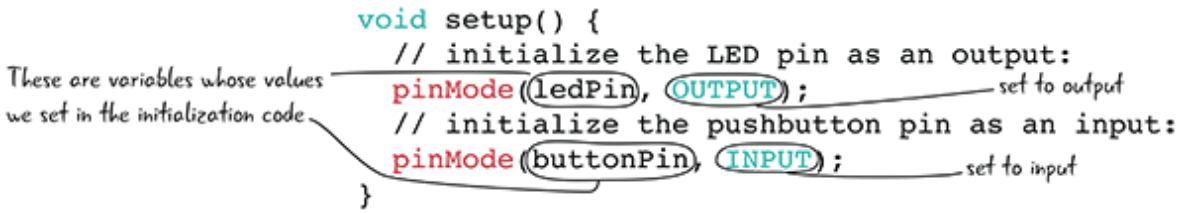
A: The error console in the Arduino IDE will give you an "unexpected unqualified id" warning, which will be displayed in orange text. The easiest fix is to change your variable name to something different.

Q: Do I need to use `const` for all my variables?

A: No, it depends on what the variable is for. For example, in the LEA6_Button sketch, the pin variables will never change during the sketch, but the variable for `buttonState` will. We want to be able to change the value of the `buttonState` variable, so we will leave `const` off its declaration.

SETUP() FOR LEA6_BUTTON

Now that we've looked at the LEA6_Button sketch's initialization code, let's move on to its `setup()` function:



The `setup()` function for the LEA6_Button sketch has only two lines of code. Similar to the code in Chapter 4, you're setting a pin to be an output using the `pinMode()` function. This time, however, you're using the variable `ledPin` to stand in for the number 13. You're also using `pinMode()` to set a different pin with value `buttonPin` as an

input. These are two of the three variables that you created in the sketch's initialization code. Naming your variables gives you a way to refer to numbers you need in your sketch in a meaningful way and makes your code easier to read.

Let's take a closer look at what we mean by digital input.

DIGITAL INPUT REFRESHER

Let's say that you're arriving at a friend's house when your friend calls and asks you to look through the window at a light. Your friend then asks, "Is the light on?" Your job is to tell the friend, "Yes, the light is on" or "No, the light is off." That's exactly what a digital input does: it reports whether the light is on or off (Figure 6-15).

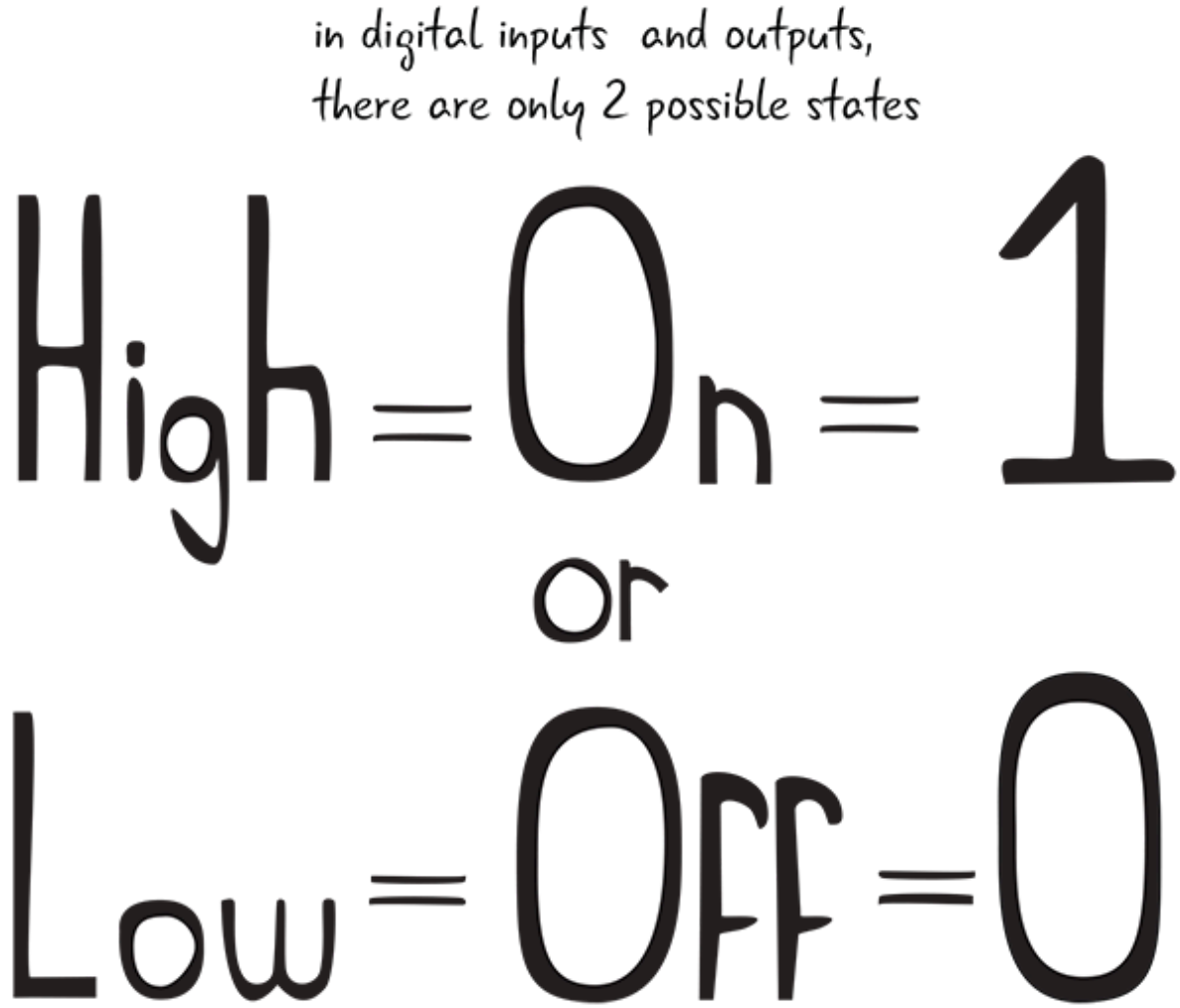


FIGURE 6-15: Digital input states

In digital inputs, there are only two possible states: `HIGH` and `LOW`, which you can think of as on (`HIGH`) or off (`LOW`). Digital inputs measure whether something is on (in a `HIGH` state) or off (in a `LOW` state). `HIGH`/on is also equal to 1 and `LOW`/off is equal to 0. We can use the digital pins on the Arduino to check on buttons and switches to see whether or not they have been triggered or pressed.

WHY THREE DIFFERENT WAYS TO SAY THE SAME THING?

If `HIGH`, 1, and on are all equivalent (as are `LOW`, 0, and off), why are there multiple ways to say the same thing? This can be confusing.

Each value talks about a different aspect of our Arduino project:

On and *off* refer to what we see happening in the world. For example, is the LED lit or not? We don't use the terms *on* and *off* in our code for the Arduino—only in our general discussions.

1 and *0* are integer variable values that represent on and off, respectively. We use 1 or 0 when we are initializing variables in our code. You saw an example of this in our sketch's initialization code, which includes the line `int buttonState = 0;`. This line tells the Arduino that the button is initially off.

`HIGH` and `LOW` refer to the electrical state of the pin: is the pin providing 5 volts or is it acting as 0v (ground)? In the Arduino programming language, `HIGH` and `LOW` are used to set or to read the state of a pin (via `digitalWrite()` and `digitalRead()` functions).

1's and 0's are part of the binary language that computers speak. `HIGH` and `LOW` means 1 and 0 to computers, including our Arduino. `HIGH` and `LOW` make the code slightly easier for humans to read, and they are used when we are using the `digitalWrite()` and `digitalRead()` functions. You'll use 0 and 1 when you are creating new variables.

LOOKING AT THE SKETCH: CONDITIONAL STATEMENTS

Now that you understand what digital inputs do, let's take a look at the LEA6_Button sketch's `loop()` code.

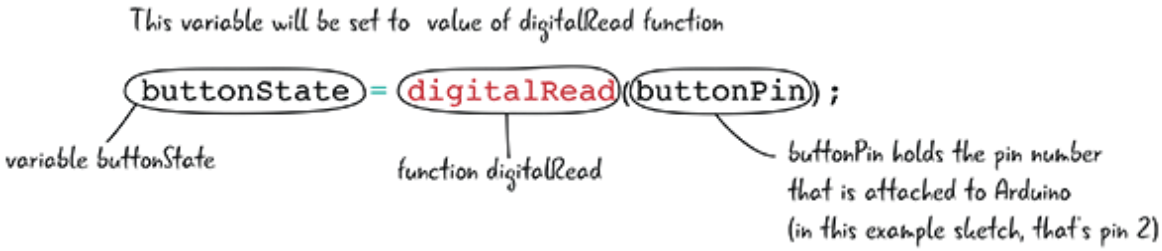
this code involves some new concepts, explained shortly

```
void loop() {
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

EXPLORING THE *LOOP()*

In the first line of the LEA6_Button sketch's `loop()` section, the Arduino uses a function called `digitalRead()` to check whether a pin is on or off. In this case, you are checking the pin represented by your `buttonPin` variable, so you are evaluating the state of Pin 2. The results of your `digitalRead()` function will be either a value of 1 (`HIGH`) or 0 (`LOW`). You then set your variable named `buttonState` to this value.



The next part of the `loop()` code gives you another new programming concept: the use of *conditional statements*.

WHAT IS A CONDITIONAL STATEMENT?

Conditional statements are a powerful way to change what happens within your code depending on conditions you specify, such as whether a button is on or off. You have experienced the use of conditional statements in everyday language, as shown in Figure 6-16.

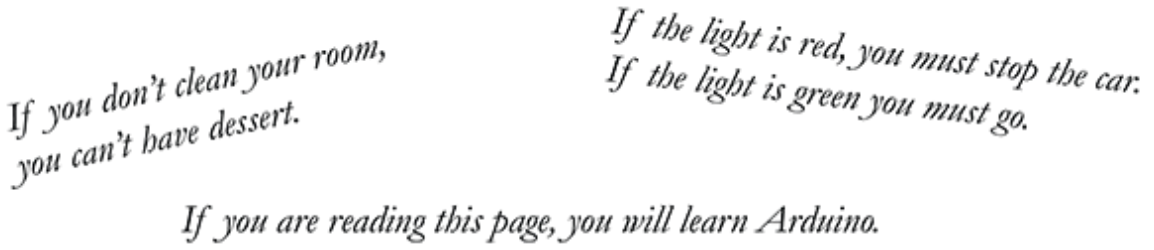


FIGURE 6-16: Conditional statements in English

Conditionals in programming work the same way. They have three basic parts: the `if`, the expression you are evaluating, and what you want to happen if our statement is true. Let's take a look at conditional statements in our `loop()` code.

The first part of the conditional statement within the `loop()` code for LEA6_Button is shown here. In some sketches, this could be your whole conditional statement; ours happens to have a second part—you'll learn about it soon.

conditional statement part one

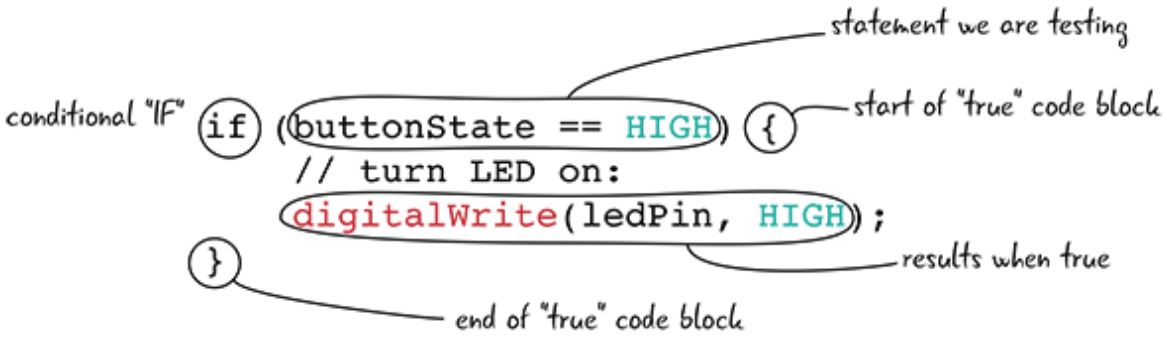
```
if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
}
```

This part of the conditional statement includes the `if`, the expression you are evaluating, and what happens if your expression

is true. Everything that will happen, if the statement is true, is contained within a set of brackets. You, as the programmer, get to tell the program what to do if certain situations happen.

CONDITIONAL STATEMENTS IN *LOOP()*

Conditional statements start with an `if`. The `if` tells the computer to evaluate the next expression.

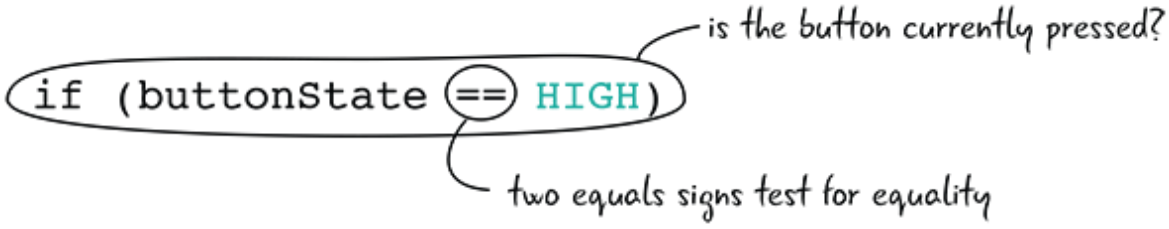


The next part of the conditional statement is the *condition to be evaluated*. This is a section of code that the Arduino has to assess for truth. "True" in a programming context means that the condition is logically valid. For example, the English statement "One is equal to one" doesn't tell us anything interesting, but it is true. The nonsensical statement "Two plus two equals five" is false. You will see various types of conditional statements throughout the book that evaluate what is happening with your Arduino and the rest of the circuit.

In the case of this sketch, the code is trying to evaluate whether the button is currently pressed. (Remember, pressed means "on," which is the same as `HIGH` in the Arduino programming language.) To test whether a value is equal to another value, you use two `=` signs, or `==`.

Note

Conditional statements start with an `if`.



The last part is the "true" code block, the commands that are run if the condition is true. There is no limit to the number of actions you can include inside the true code block, as long as they are all contained within the brackets. In this case, the code block will turn on the LED attached to the `ledPin`, also known as Pin 13.

```
if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
}
```

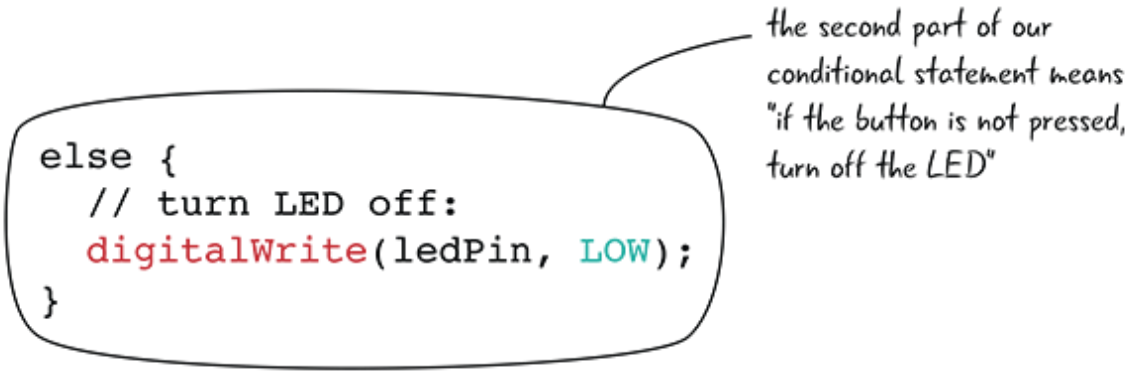
> **Note**
>
> Conditional statements check for whether something is logically true.

> **Tip**
>
> Think carefully about what you want your conditional statement to do. You might try saying it out loud to yourself.

CONDITIONAL STATEMENTS: *ELSE*

What happens if the button is *not* pressed? For this conditional statement, there is an `else` clause, which handles any events that happen when the statement is *not true*. `else` is helpful for dealing with cases where the `if` statement is false, but it is not required for every conditional statement. Some conditionals have an `else`, and some do not. If this conditional didn't have an `else`, then if the condition you're evaluating were false, nothing would happen.

For your button code, the `else` statement can also be broken down into a simple English statement: "If the button is not pressed, then turn off the light."



Note

`else` is not required in all conditional statements.

Table 6-1 shows a quick summary of the conditional statement we just examined.

TABLE 6-1: Conditional statement in LEA6_Button

WHAT IS HAPPENING IN THE CIRCUIT?	CONDITION TO BE EVALUATED	TRUTH VALUE	RESULT
Button is pressed	`if (buttonState == HIGH)`	true	Turns LED on
Button is not pressed	`if (buttonState == HIGH)`	false	Turns LED off

QUESTIONS?

Q: What if I want more than two possible outcomes?

A: Then you might use an `else if`, or maybe even multiple `else if`s. You can read more about it here: arduino.cc/en/Reference/Else.

Q: Can I place a conditional inside another conditional?

A: Yes, it is possible to have conditionals inside other conditionals. Although you won't see an example in this book, they are called nested conditionals, and they can let you deal with evaluating complicated logic.

Now that you've connected your button and made it turn the LED on and off, you are ready to make your circuit more interesting. Let's add a speaker, and then add some code so that the speaker plays a tone when you press the button. First we'll show you how to add the speaker to the breadboard.

ADD A SPEAKER AND ADJUST THE CODE

In this circuit, the button, LED, resistors, and jumpers will stay in the same place. You are simply adding a speaker; everything else remains the same (Figure 6-17).

Part to add:

1 8-ohm speaker

As always, before you attach the speaker, make sure your computer is not connected to the Arduino. Attach one end of the speaker to Pin 11 on the Arduino and the other end to the ground

bus (Figure 6-18). It doesn't matter which end you connect where; like a resistor, the speaker does not have an orientation. The colors of your speaker wire may vary, but the speaker does not have a direction.

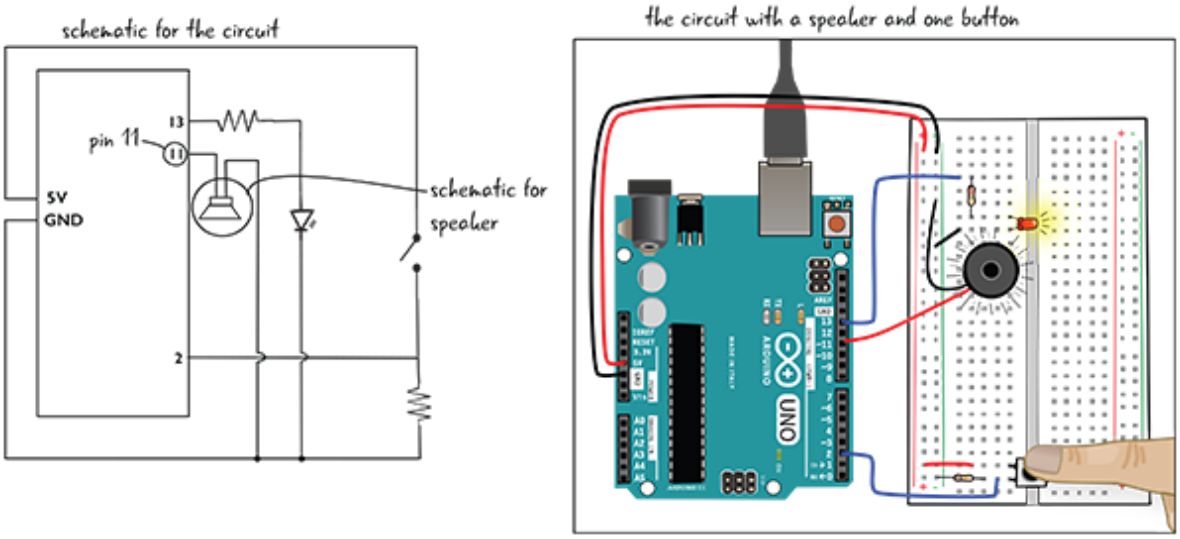


FIGURE 6-17: A speaker added to the circuit

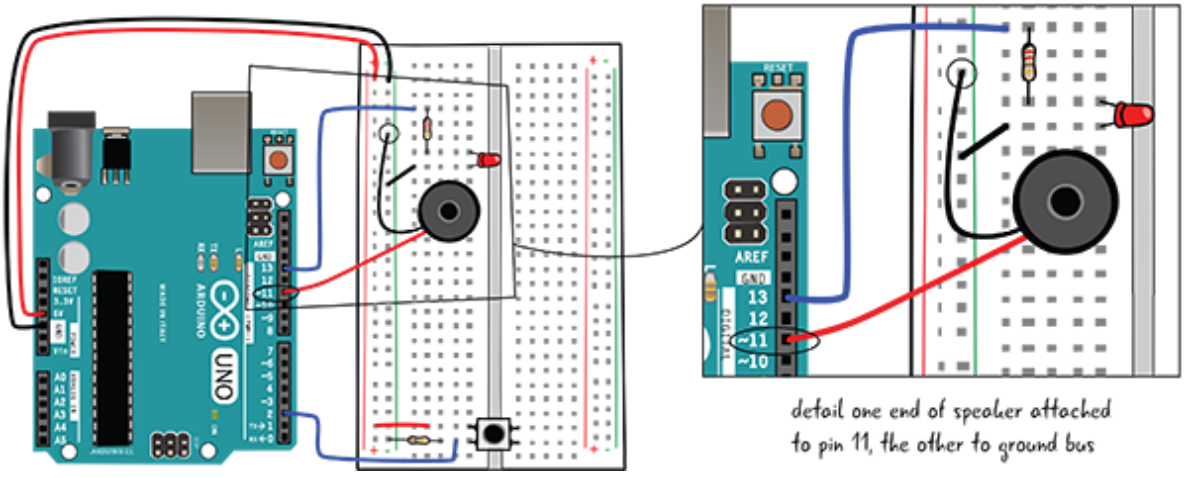


FIGURE 6-18: Adding the speaker

That's all there is to adding a speaker. Now you are ready to adjust your code.

ADDING CODE FOR THE SPEAKER

Now that you've wired up the speaker, you will adjust the code. First, save the sketch as a new sketch named LEA6_1_tonebutton.

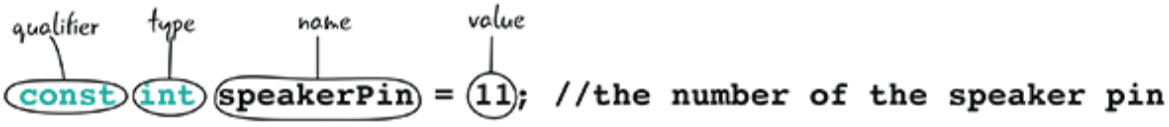
You're going to add a line of code to the sketch's initialization section and add a variable for the speaker pin.

```
const int buttonPin = 2;   // the number of the pushbutton pin
const int ledPin = 13;      //the number of the LED pin
const int speakerPin = 11; //the number of the speaker pin

// these variables will change:
int buttonState = 0;
```

add this variable to hold value of the speaker pin

Let's look at the new line of code more closely. You can see that it is like your other variable declarations: there is a qualifier, a type, a name, and a value. Remember, you use the `const` qualifier, which stands for constant, when you have a variable that has a value that will not change.



> **Tip**
>
> It's a good idea to add comments as you type new code to remember what you are adding to your sketch.

ADJUSTING *SETUP()*

What do you think you will have to adjust in the sketch's `setup()` section? Remember that `setup()` is where you state whether the circuit's various components are inputs or outputs.

What is the speaker? An output. So you'll add a line of code that will declare that the pin the speaker is attached to is an output. Because you made a variable to hold that value, you will use it when you declare the pin an output.

```
void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
  pinMode(speakerPin, OUTPUT);
}
```

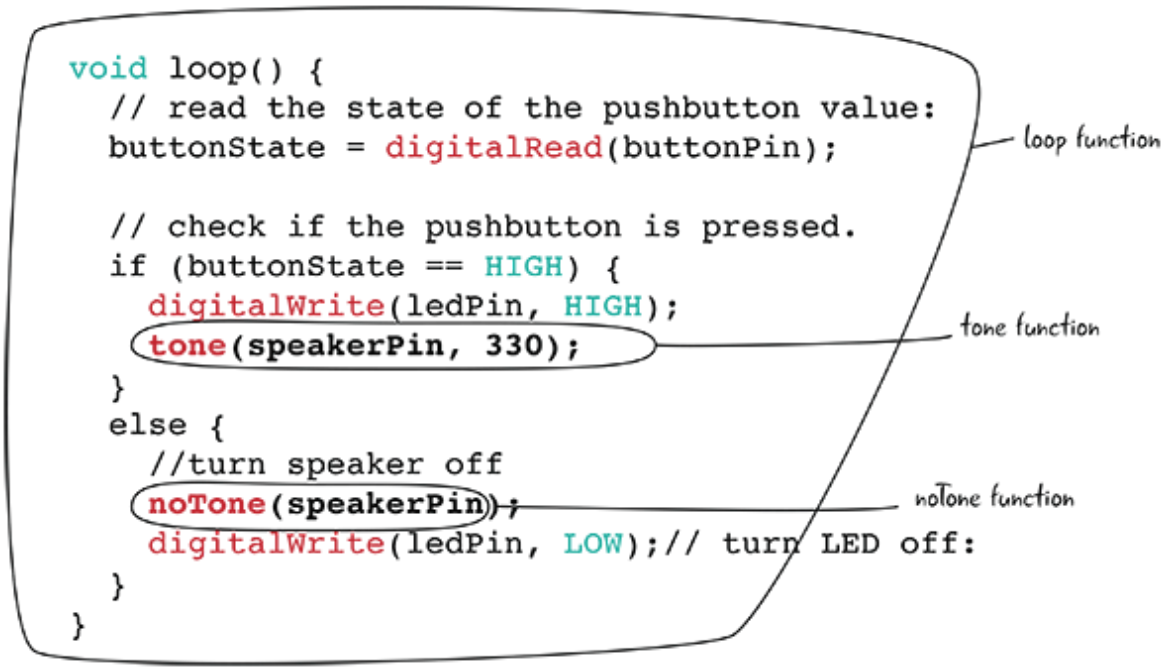
use the pinMode function to declare speakerPin an output

Here's a closer look at this new line of `setup()` code:

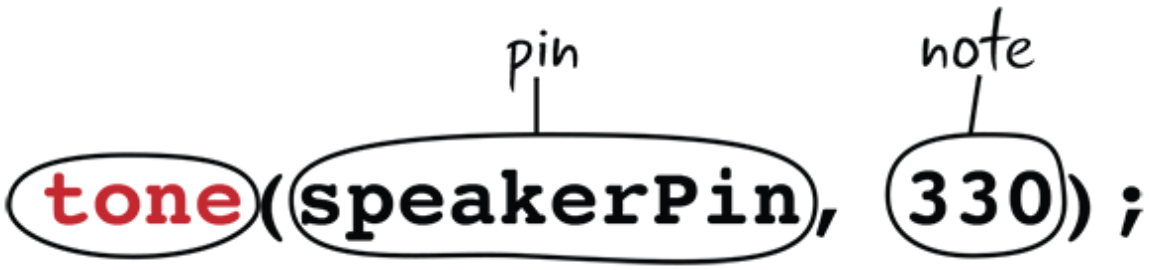
On to `loop()`!

ADJUSTING *LOOP()*

As you have seen, `loop()` has the code that reads whether or not the button is pressed and then uses a conditional to direct the Arduino to do something based on that information. Now you will use the Arduino functions `tone()` and `noTone()` inside the conditional. `tone()` will generate a note or tone; `noTone()` will stop it from being played. Let's look at all of the `loop()` code first, and then explore `tone()` and `noTone()` more closely.



As we said, `tone()` will generate a note or tone that can play through the speaker you just attached. When you use the `tone()` function, you need to tell Arduino on which pin to generate a tone and what note to play. It makes sense that you want to generate a note on the pin that has the speaker attached.

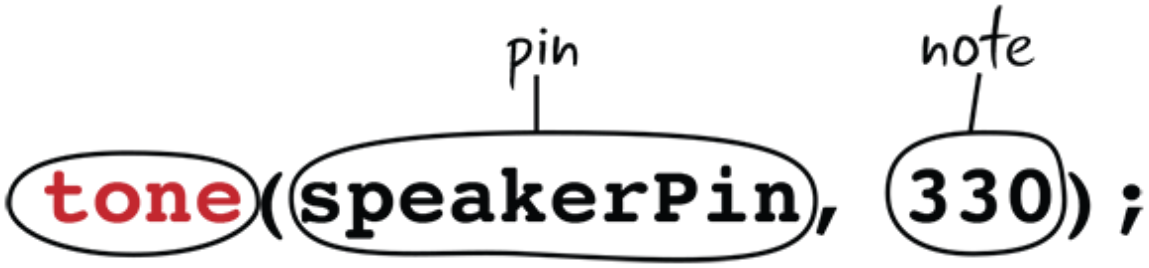


Let's take a more in-depth look at the `tone()` and `noTone()` functions.

TONE() AND NOTONE() UP CLOSE

What does `330` mean? You know it means the note the speaker will play, but how do you arrive at that number? The Arduino generates

sound waves that are measured in hertz; 330 is the hertz value of the note you want your circuit to play.



In musical circles, this note is known as an E. From now on, when we mention the `tone()` function, we will say that it is generating a note. Figure 6-19 shows some of the possible note values.

the values for a few notes

NOTE	FREQUENCY (hertz)
C	262
D	294
E	330
F	349
G	392
A	440
B	494
C	523

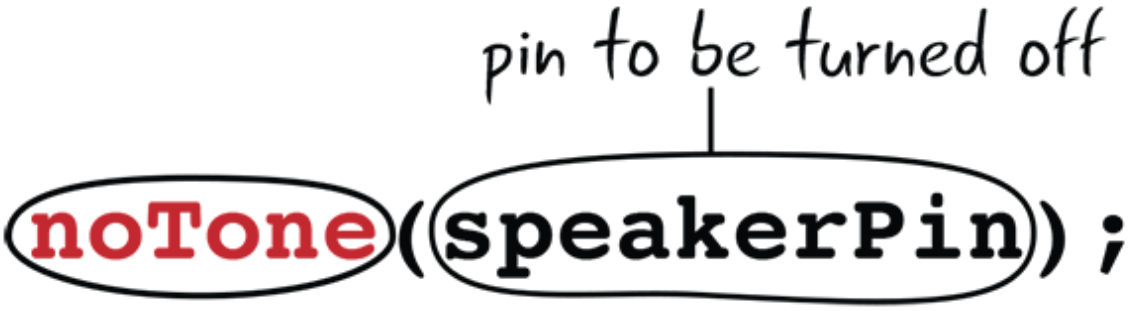
our first note

the sound an orchestra tunes to

FIGURE 6-19: Note chart

> **Tip**
>
> A more comprehensive note chart can be found on the Arduino website at arduino.cc/en/Tutorial/ToneMelody.

Now let's look at `noTone()`. This function stops the sound from being played on the pin specified. In this case, that's `speakerPin`, which stores the value of the pin that has the speaker attached to it. If you leave out `noTone()`, then your note will play continuously once you press the button the first time.



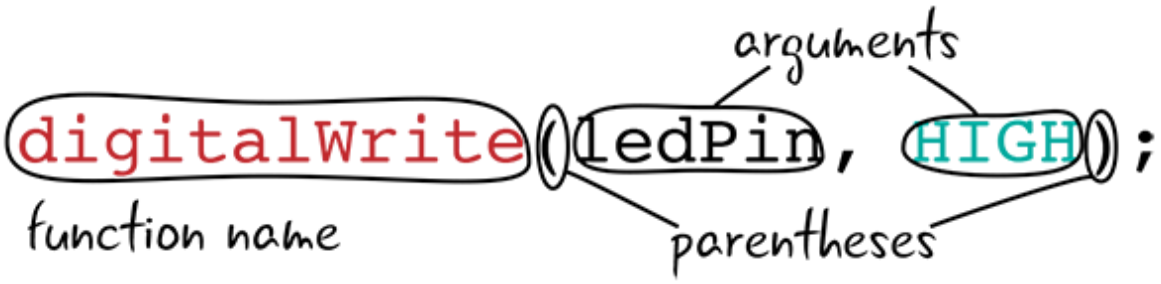
Add the `tone()` and `noTone()` functions to your code, save your sketch, and then upload. Now when you press the button, you'll hear a note playing from the speaker, as well as see the LED turn on.

We said earlier that we would explain what's inside the parentheses in functions. For example, in the `tone()` function, what do `speakerPin` and `330` mean? Those values are called *arguments*. Let's take a look at them now.

ARGUMENTS

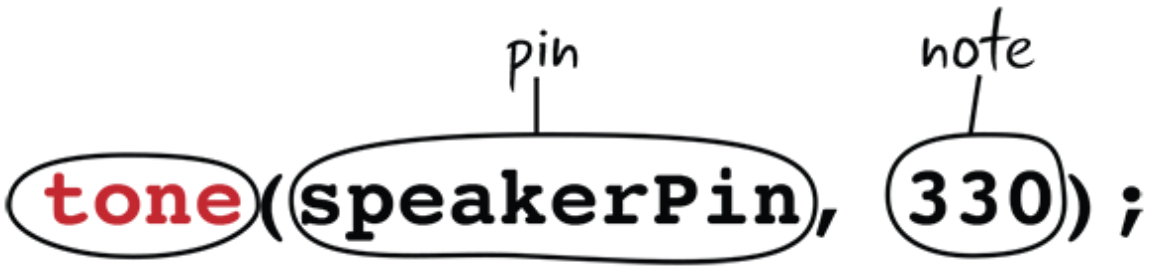
You've used a number of Arduino functions in this book so far, from `pinMode()` to `digitalWrite()`. You may have also noticed that most of the functions require something to be placed within the

parentheses, often some combination of numbers and words. The values placed inside the function are called *arguments*.

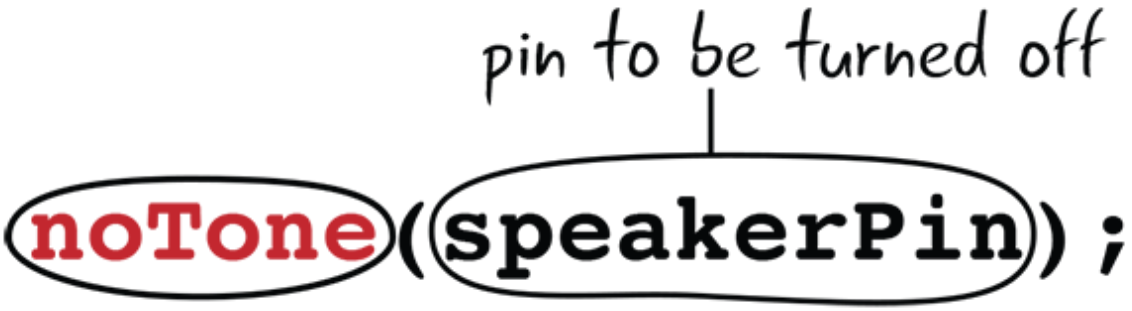


Arguments tell your Arduino function important information, such as which pins are used as inputs. Different functions often have a different number of arguments. `digitalWrite()` has two arguments —the pin number and the value—whereas the Arduino `delay()` function has only a single argument—how many milliseconds to pause the program. Some functions won't need any arguments, whereas others will require several. Let's look at the `tone()` and `noTone()` functions and how arguments work with them.

In the `tone()` function, two arguments are passed in: the pin that the speaker is attached to (in this case, the variable `speakerPin`) and the value of the note. Note that the argument values are separated by a comma.



The `noTone()` function has one argument: the pin that the speaker is attached to (again, the variable `speakerPin`).



Some functions have no arguments, whereas others have many. In some functions, not all of the arguments are required. You'll learn more about functions and arguments in later chapters.

Next, you'll add a second button to your tone button keyboard so you can play more than one note.

ADD TWO MORE BUTTONS AND ADJUST THE CODE

You are going to add another button to your circuit so you can play a two-note tune on your mini-keyboard instrument (Figure 6-20). You will need another button, another 10 kΩ resistor, and more jumpers. Remember to unplug your Arduino from your computer before you add to the circuit.

Parts to add:

1 momentary pushbutton switch

1 10 kΩ resistor (brown, black, orange, gold)

Jumper wires

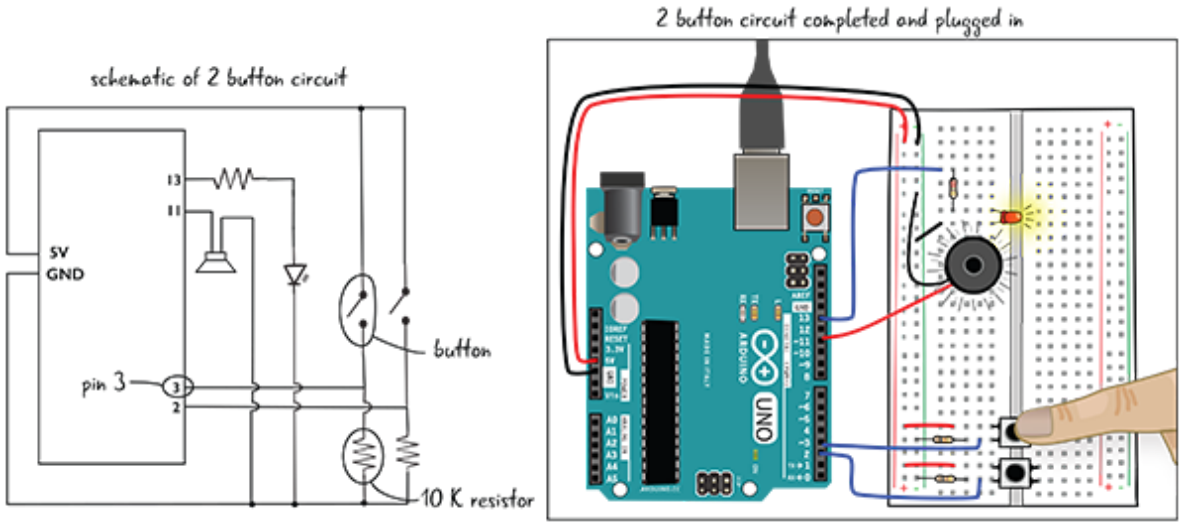


FIGURE 6-20: Two-button circuit

The configuration of this button will be very similar to the first button you placed in the circuit, except that the new button will be attached to a different pin on the Arduino (Figure 6-21).

Place the new button across the trench. Use a jumper wire to connect the button's top-left pin to the power bus. Attach one lead of the 10 kΩ resistor to ground and the other lead to the lower-left pin of the button. Finally, attach Pin 3 to the lower-left pin of the button and one end of the 10 kΩ resistor with a jumper wire.

Now that you've added the second button, it's time to adjust the code in the sketch.

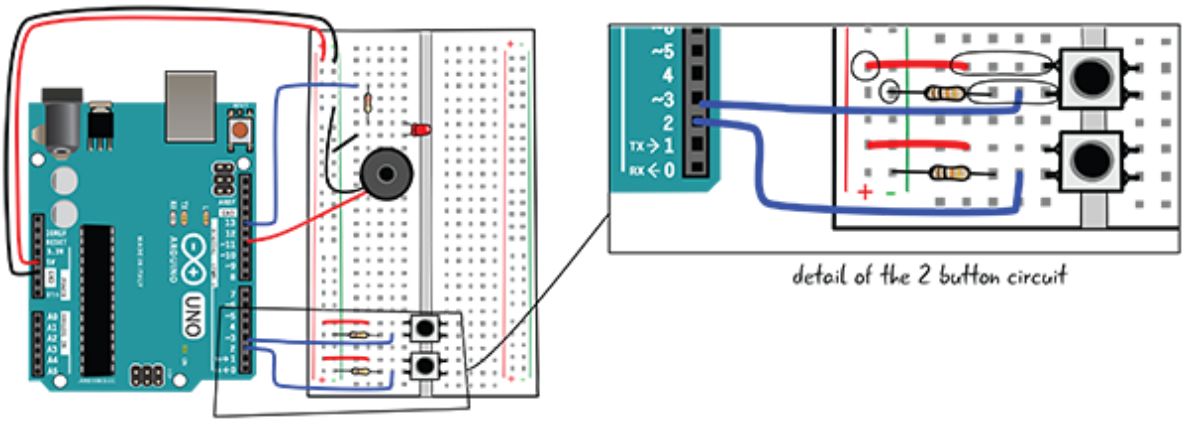


FIGURE 6-21: Attaching the second button

EDITING LEA6_2_TONEBUTTONS

First, save your sketch as LEA6_2_tonebuttons. You will edit your code, adding the lines that are marked in bold on the next couple of pages.

Initialization Code Adjustments

Here is the initialization code updated with two new variables. One is set to the number of the pin you attached to the second button (3); the other will hold the state of that button, and it is initially set to 0.

```
const int buttonPin = 2;    // the number of the pushbutton pin
const int buttonPin2 = 3;   // a second pushbutton pin
const int ledPin =  13;     // the number of the LED pin

// variables will change:
int buttonState = 0;        // variable for reading the pushbutton status
int buttonState2 = 0;       // variable holds second pushbutton state
```

variable set to number of pin attached to the second button

this variable will hold the state of the second button, as a 0 or 1 to indicate whether it is being pressed

setup() Code Adjustments

The following graphic shows the `setup()` code again, edited to account for your second button. You use the `pinMode()` function again, this time to set `buttonPin2` (which you set to 3 in the initialization code) as an input.

```
void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pins as inputs:
  pinMode(buttonPin, INPUT);
  pinMode(buttonPin2, INPUT);
  pinMode(speakerPin, OUTPUT);
}
```

Next, let's look at the `loop()` code and see what you need to add.

ADJUSTING THE *LOOP()* CODE: *ELSE IF*

You can see that you are reading the value of `buttonPin2` (either 1 for on or 0 for off) with the `digitalRead()` function and storing it in the variable `buttonState2`.

You also have to add a new section inside your `if` statement—what is known as an `else if`. When your `if` statement is being evaluated, it will check for the first condition after the `if` to see whether that condition is true or false. As you saw earlier, if the first condition is true (in other words, if button 1 is currently pressed), then the speaker will play a note at 330 hertz. But if the first condition is false (in other words, if button 1 is *not* currently pressed), then the Arduino will move on to the `else if` code to determine whether the statement following the `else if` is true or false. If it is true (in other words, if button 2 is currently pressed), then the Arduino will follow the instructions inside the curly braces.

```
void loop() {
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);
  buttonState2 = digitalRead(buttonPin2);
  // check if the pushbutton is pressed.
  if (buttonState == HIGH) {
    digitalWrite(ledPin, HIGH);
    tone(speakerPin, 330);
  }
  // check if the second button is pressed
  else if (buttonState2 == HIGH) {
    digitalWrite(ledPin, HIGH);
    tone(speakerPin, 294);
  }
  else {
    noTone(speakerPin); //turn speaker off
    digitalWrite(ledPin, LOW); // turn LED off:
  }
}
```

reading state of buttonPin2 and storing it in buttonState2

else if tests for second condition

Let's look at each line of the `else if`.

First, `else if` tells us that the Arduino is going to test for another condition. It is testing whether `buttonState2` is `HIGH`—in other words, is the button at Pin 3 currently being pressed?

```
else if (buttonState2 == HIGH) {
```

else if statement

condition to be tested

curly brace marks off beginning of else if block of code

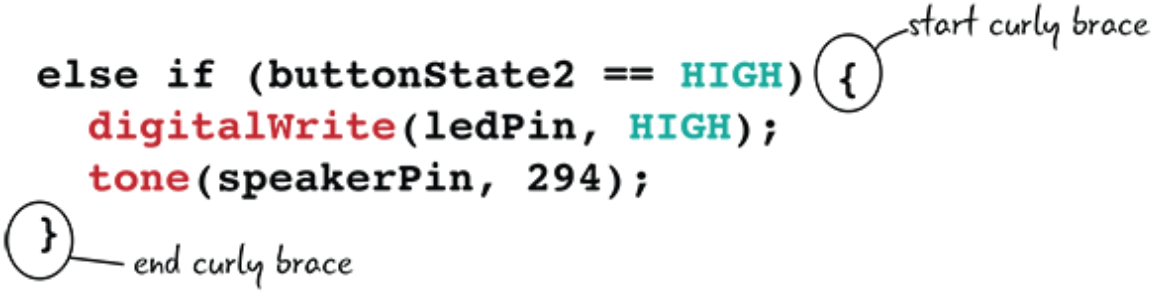
You've seen the code in the next line of the `else if` block before; it sets the pin attached to the LED to `HIGH` so that the LED turns on.

```
digitalWrite(ledPin, HIGH);
```

And here's the last line in the `else if` block. It uses the `tone()` function to play a note on the speaker. This time, the note is 294 hertz—slightly lower than the note played by the first button.

```
tone(speakerPin, 294);
```

The following graphic shows the whole block of the `else if` code again. Note that the parentheses surround the code that describes the condition being tested, and the curly brackets surround what you want to do if the condition is true.



To test your code, attach your computer to your Arduino, save your code, verify it, and upload in to the Arduino.

Now you can play two different notes on your two-button keyboard.

QUESTIONS?

Q: What will happen if I push both buttons at once?

A: The way we have written the conditional statement, only the first note will play if you push both buttons at once. This works in our favor because the Arduino `tone()` function isn't able to play more than one tone through the speaker at a time.

Q: What will happen if I change the second note number in the `tone()` function to something other than 294?

A: The note chart you saw a few pages ago provides just a selection of the possible notes you can play. We left out all sharp/flat notes—this isn't a lesson on music theory—but if you randomly pick a number for the second note, chances are it will sound slightly off—like an out-of-tune guitar—compared to the first note.

ADDING THE THIRD BUTTON

Now you're going to add a third and final button to the circuit (Figure 6-22). Be sure to unplug your Arduino from your computer before adding this button!

Parts to add:

1 momentary pushbutton switch

1 10 kΩ resistor (brown, black, orange, gold)

Jumper wires

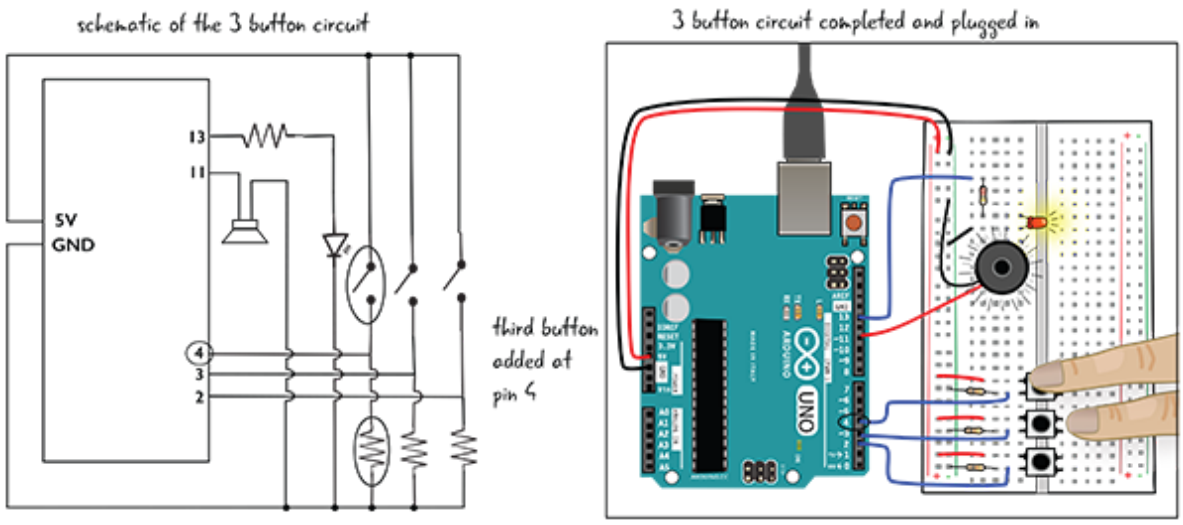


FIGURE 6-22: Three-button circuit

Place the button across the trench above the other two buttons. Use a jumper to connect the button's top-left pin to the power bus. Attach one lead of the 10 kΩ resistor to ground and the other lead to the lower-left pin of the button. Finally, attach Pin 4 to the lower-left pin of the button and one end of the 10 kΩ resistor (Figure 6-23).

Now that you've added the third button to the circuit, it's time to adjust the sketch.

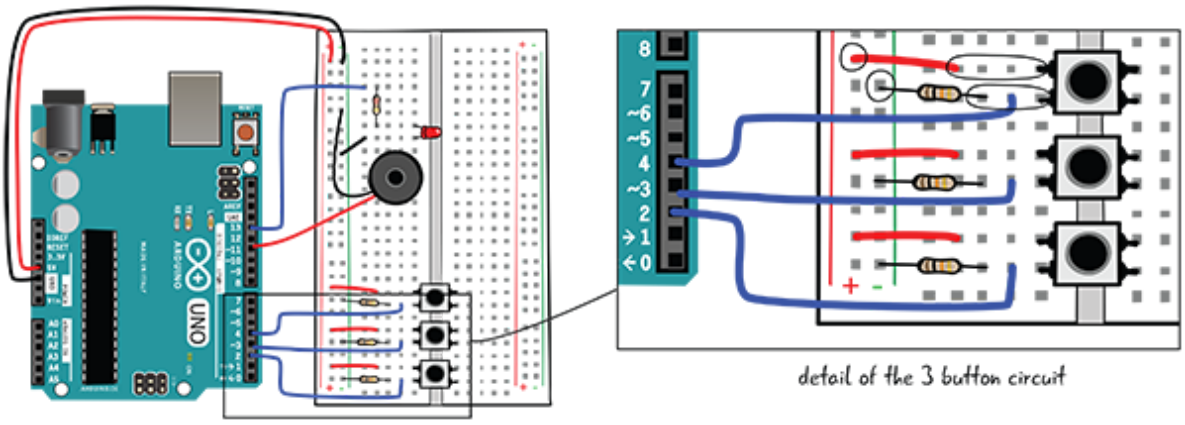


FIGURE 6-23: Adding the third button

EDITING THE LEA6_3_TONEBUTTONS SKETCH

Save your sketch as LEA6_3_tonebuttons, and we'll look at the code you have to adjust. This will be very similar to the adjustments you made to the previous sketch when you added the second button. Follow along and edit your sketch to match the following code.

Initialization Code Edits

You are attaching the third button to Pin 4. You're also adding a variable called `buttonState3`, which holds the value that indicates whether or not the pin is being pressed (1 or 0, `HIGH` or `LOW`, on or off).

```
const int buttonPin = 2;    // the number of the first pushbutton pin
const int buttonPin2 = 3;   // a second pushbutton pin
const int buttonPin3 = 4;   // third pushbutton pin attached to pin 4
const int ledPin =  13;     // the number of the LED pin
const int speakerPin = 11; // the number of the speaker pin

// variables will change:
int buttonState = 0;        // variable for reading the pushbutton status
int buttonState2 = 0;       // variable holds second pushbutton state
int buttonState3 = 0;       // third pushbutton state
```

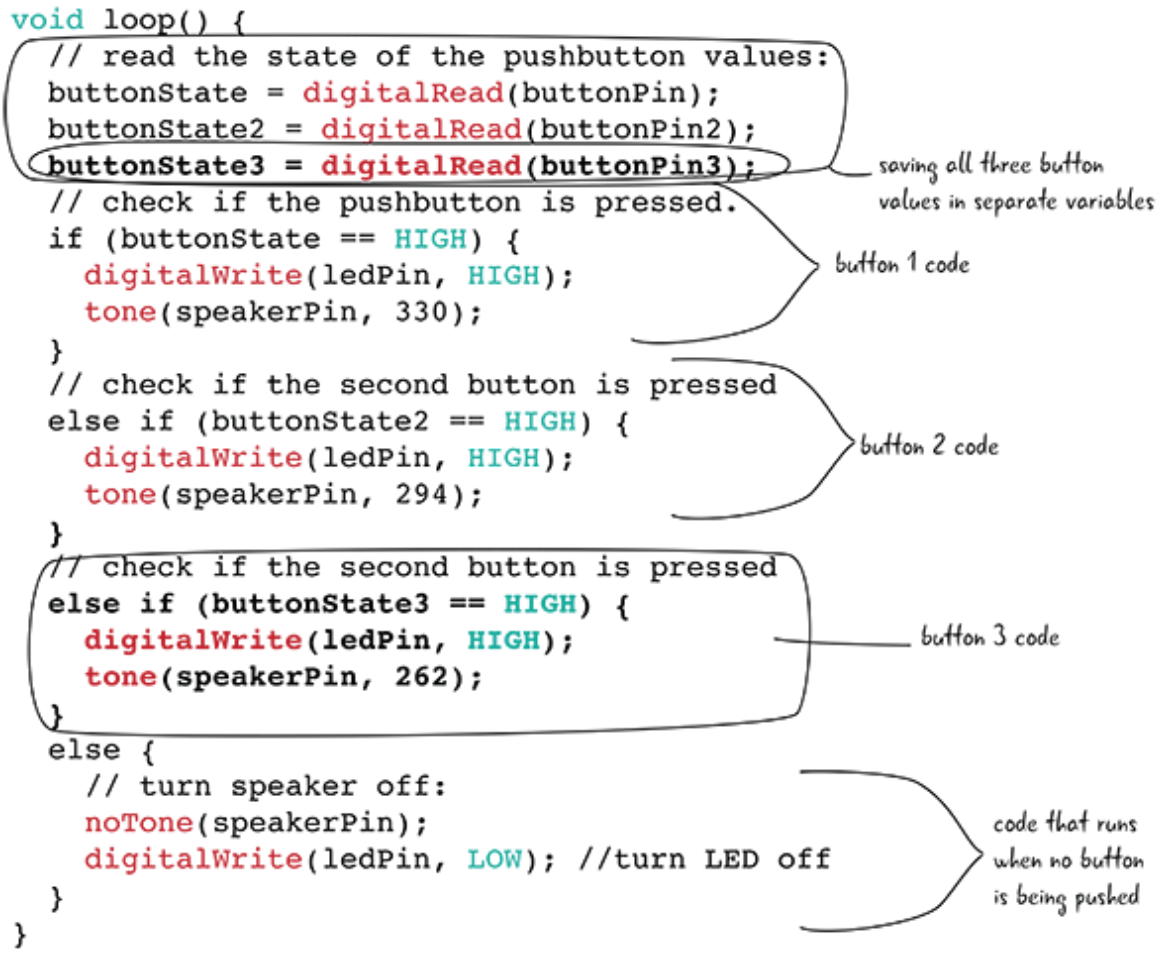
setup() Code Edits

In our `setup()` code, we set the variable `buttonPin3` (which holds the value 4 to indicate pin 4) to an `INPUT`. All three of our buttons have now been set as `INPUT`s.

```
void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pins as inputs:
  pinMode(buttonPin, INPUT);
  pinMode(buttonPin2, INPUT);
  pinMode(buttonPin3, INPUT);
  pinMode(speakerPin, OUTPUT);
}
```

Three-Button "Instrument" *loop()* Function

The updated `loop()` function is shown next. It reads the state of `buttonPin3` and stores it in `buttonState3`. It also has an additional `else if` statement, which tests to see whether the third button is being pressed, and if so, plays a note (one slightly lower than the note for button 2) and lights the LED.



Your code can now respond when you press each of the three buttons.

PLAY YOUR MINI-KEYBOARD INSTRUMENT

You've written your code and adjusted your circuit. Your three-button mini-keyboard instrument should now work. To take it for a spin, attach your computer to your Arduino, save your code, verify it, upload it to the Arduino, and then press the buttons. Remember to press one button at a time, since the speaker can play only one note at a time.

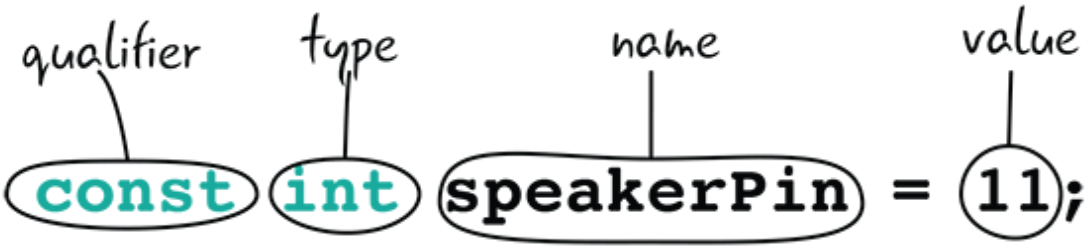
Before we move on to look at how some of the components are working in this circuit, we'll briefly review what you've learned about writing code while building this project.

REVIEWING ELECTRONIC AND CODE CONCEPTS

You learned a few new, and very important, programming concepts in this chapter. These concepts are critical to writing code in all programming languages, though the details might be a little different depending on the language. Let's look once more at variables and conditionals.

VARIABLES

A variable is a container in your code that can hold different values.



CONDITIONAL STATEMENTS

A conditional statement evaluates a condition and executes instructions if that condition is true. If the conditional statement contains an optional `else if` or `else` block, then it can test for multiple conditions, and it sometimes tells the code to do something if the conditions are not true.

```
if (buttonState == HIGH) {
  digitalWrite(ledPin, HIGH);
}
// check if the second button is pressed
else if (buttonState2 == HIGH) {
  digitalWrite(ledPin, HIGH);
}
```

Let's take a quick look at how the electronic components you used in this chapter work in a circuit.

HOW DOES THE BUTTON WORK?

The default, unpressed state of the button is open, meaning electricity can't flow through it. In order for electricity to flow through your button, it must be pressed down, making a connection between the pins (Figure 6-24). When you read the value on the pin that is attached to the button, you will see that it is `HIGH`.

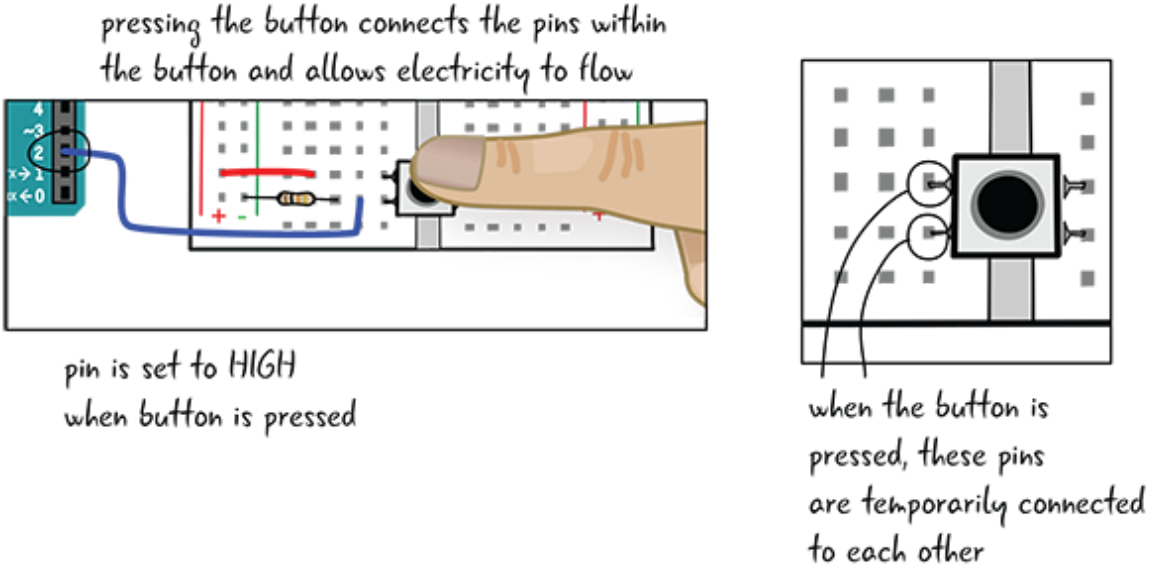


FIGURE 6-24: Pushing the button

There is nothing attached to the pins on the other side of the trench, but the pins on that side of the trench are also connected when the button is pressed (Figure 6-25).

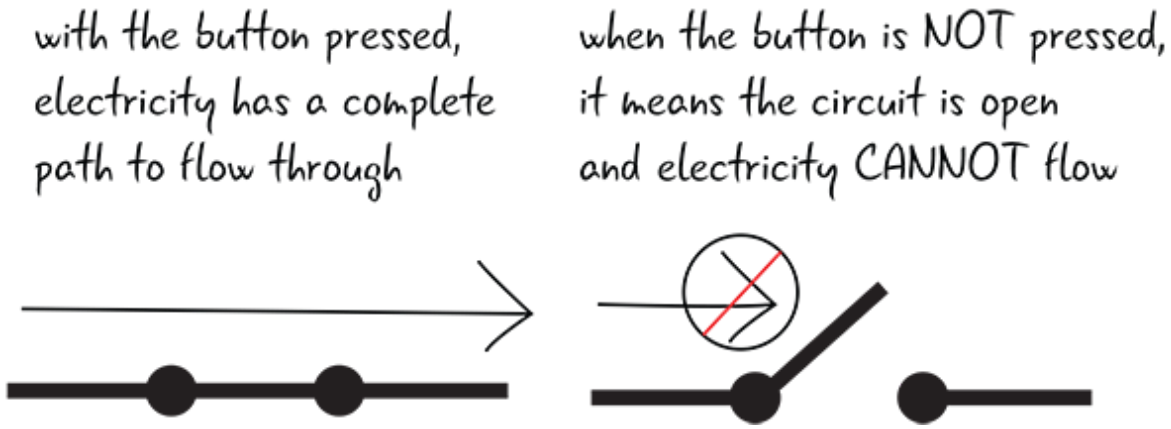


FIGURE 6-25: How a switch functions

HOW DOES THE SPEAKER PLAY DIFFERENT NOTES?

The `tone()` function built into the Arduino knows how to change the power provided by your digital pin to create different notes from your speaker. Without getting too technical, the note value you include in your `tone()` function tells the Arduino how to rapidly change the voltage to create different notes (Figure 6-26).

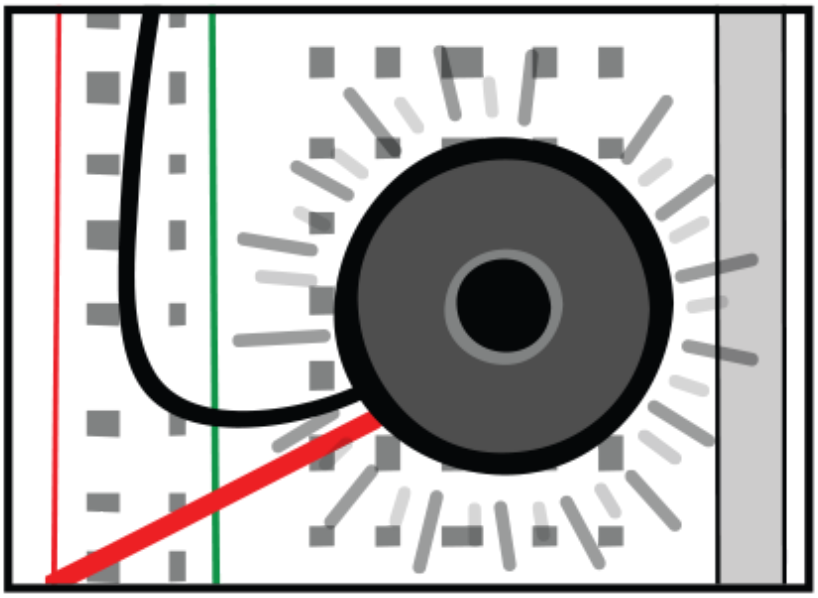


FIGURE 6-26: A change in voltage to the speaker will play different notes.

SUMMARY

This chapter taught you more about programming. You learned what a variable is and how to use it, and how to use conditional statements to control the flow of your program. You also learned more about digital output and how to add a digital input to your circuit to make your project interactive. You can download the code here: github.com/arduinotogo/LEA/blob/master/LEA6_3_toneButtons.ino.

In the next chapter, we will show you how to attach analog sensors or other inputs to a circuit and use the information you gather from them to do more with your output components than turn them on and off.

7

ANALOG VALUES

In the previous chapter, you learned how to put buttons into a circuit to play notes through a speaker and to turn an LED on and off. This chapter shows you how to attach sensors to a circuit and use the information you gather from them to create more varied experiences. You will also learn how to use the Arduino IDE to look at information coming in from your sensors.

THERE'S MORE TO LIFE THAN ON AND OFF!

You have learned how to attach buttons to your circuit so you can make your projects interactive using digital inputs and outputs with your Arduino sketches. With digital input, you have only two possible values: on or off (a.k.a. `HIGH` or `LOW`, 1 or 0). But sometimes you might want to use values that are not as simple as on or off. In this chapter, you will see how to read values from sensors and variable resistors, and then use those values in your Arduino sketches to produce different effects.

You will learn these concepts by building a circuit with a potentiometer, which is like a knob that can be turned to give you a range of values beyond just 1's and 0's. You will use your potentiometer first to adjust the brightness of an LED and then to play different notes from a speaker.

Why are we showing you how to use analog sensors and information? And what exactly do we mean by *analog*?

You have seen that digital information has only two possibilities: on and off. Analog information, on the other hand, can hold a range of possible values. We perceive the world as a stream of analog information via our sight, hearing, and other senses. By using analog information with your Arduino, you can respond to user input in a complex fashion. You can control the brightness of an LED, setting it to shine brightly, grow dimmer, or show any range of values in between.

> **Note**
>
> Analog information is continuous and can hold a range of possible values.

Once you understand how analog values work, you will use the values to create a homemade musical instrument called a *theremin*. A theremin is a musical instrument in which the pitch of the sound is controlled by the distance of the musician's hands from the instrument, as shown in Figure 7-1. (That's right; you don't actually touch a theremin to play it.) You may have heard the eerie tones of a theremin on a soundtrack for a movie or television show. Our version will use a speaker and a photoresistor; as you raise or lower your hand over the photoresistor, the speaker will play different notes.

In all of the projects in this chapter, a sketch will read information from an analog input and then use that information to control an

output, such as the brightness of an LED or the tones emanating from a speaker.

You will be using analog information in this chapter and in some of the projects in future chapters. Let's get started!

FIGURE 7-1: Playing a theremin

THE POTENTIOMETER CIRCUIT

Figure 7-2 shows the schematic and a drawing for your first circuit of the chapter. The circuit uses the potentiometer to change the brightness of your LED. The LED gets brighter as you turn the potentiometer until it is all the way lit, whereas turning it the opposite way dims the LED until it is off.

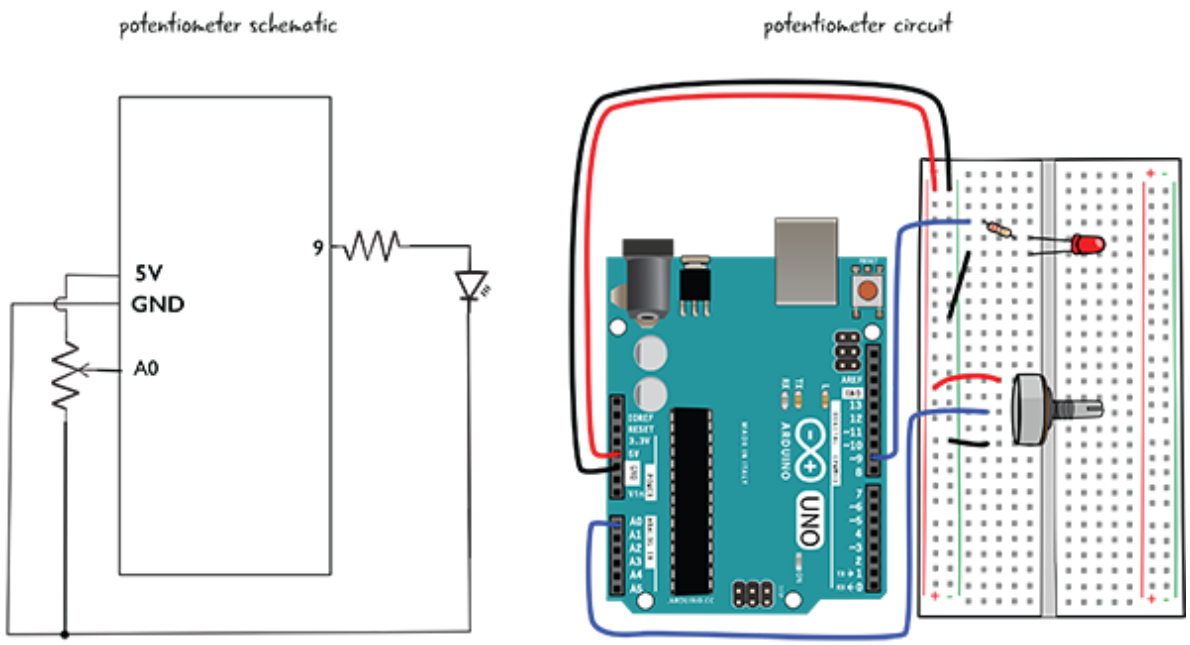


FIGURE 7-2: The first circuit you will build in this chapter

We'll discuss the analog input pins on the Arduino before we get started building our circuit.

THE ARDUINO'S ANALOG INPUT PINS

Remember back in Chapter 2, "Your Arduino," when you first took your Arduino out of the box? We pointed out that it has analog input pins, which are pins that can read sensors that have a range of possible values. Let's take a closer look at those pins in Figure 7-3.

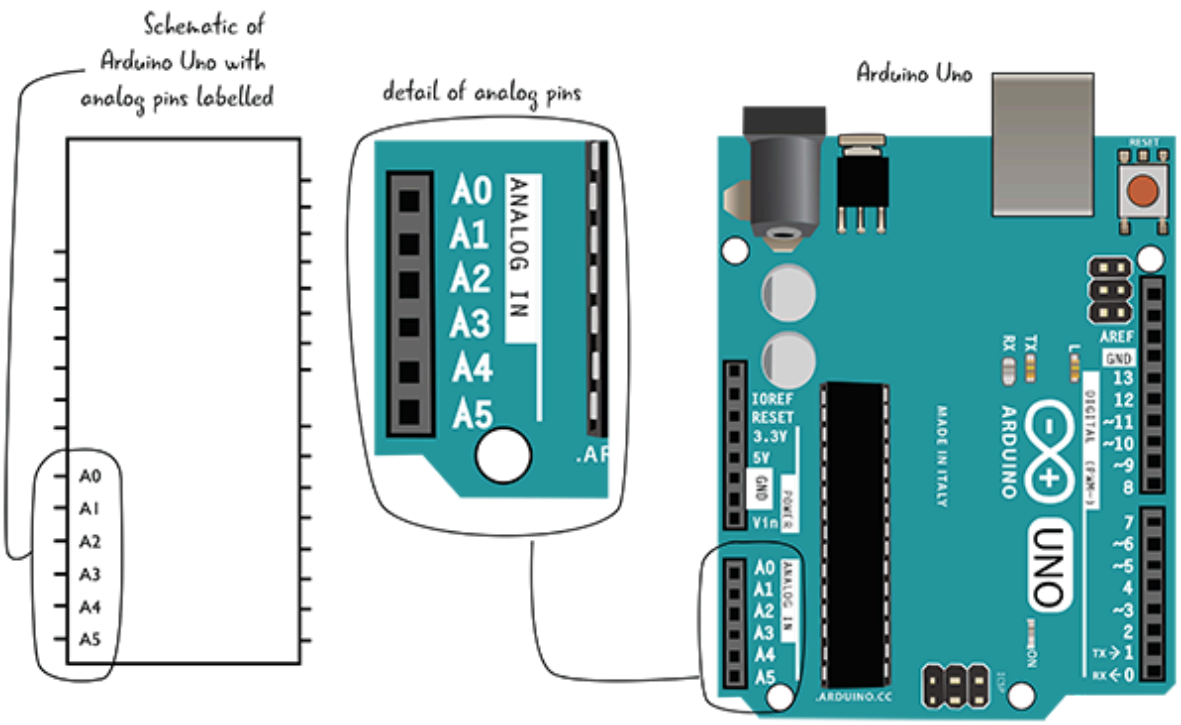


FIGURE 7-3: Analog pins on the Arduino

The analog input pins are located opposite the digital input/output pins on the Arduino, below the power and ground pins. There are six pins, labeled A0 through A5, with the "A" indicating that it is for "analog."

When one of these pins is connected to an analog input, it can return a range of values, from 0 to 1023. This range of numbers is related to how the Arduino manages memory. A detailed explanation is beyond the scope of this book; what's important for you to know is that it is a much larger range than just 1 or 0, allowing you to create varied experiences rather than simply turning something on or off.

What's an analog input? Any component, often some type of sensor, that can give you a range of values, not just on and off. The first analog input you're going to work with is a potentiometer, which you'll attach to Pin A0.

The schematic in Figure 7-3 shows the location of all of the analog pins on your Arduino.

MEET THE POTENTIOMETER

A potentiometer is a type of variable resistor, which means its amount of resistance can change. A potentiometer, sometimes called a pot, is a knob or dial that can be turned to increase or decrease the amount of resistance depending on how far, and in which direction, it is turned (Figure 7-4). Potentiometers come in many sizes and shapes. You will be using a 10 kΩ potentiometer in your circuit.

A potentiometer has three pins: one that attaches to power, one that attaches to ground, and one that attaches to a pin on the Arduino. In the following pages, we'll show you how to attach the potentiometer to a breadboard.

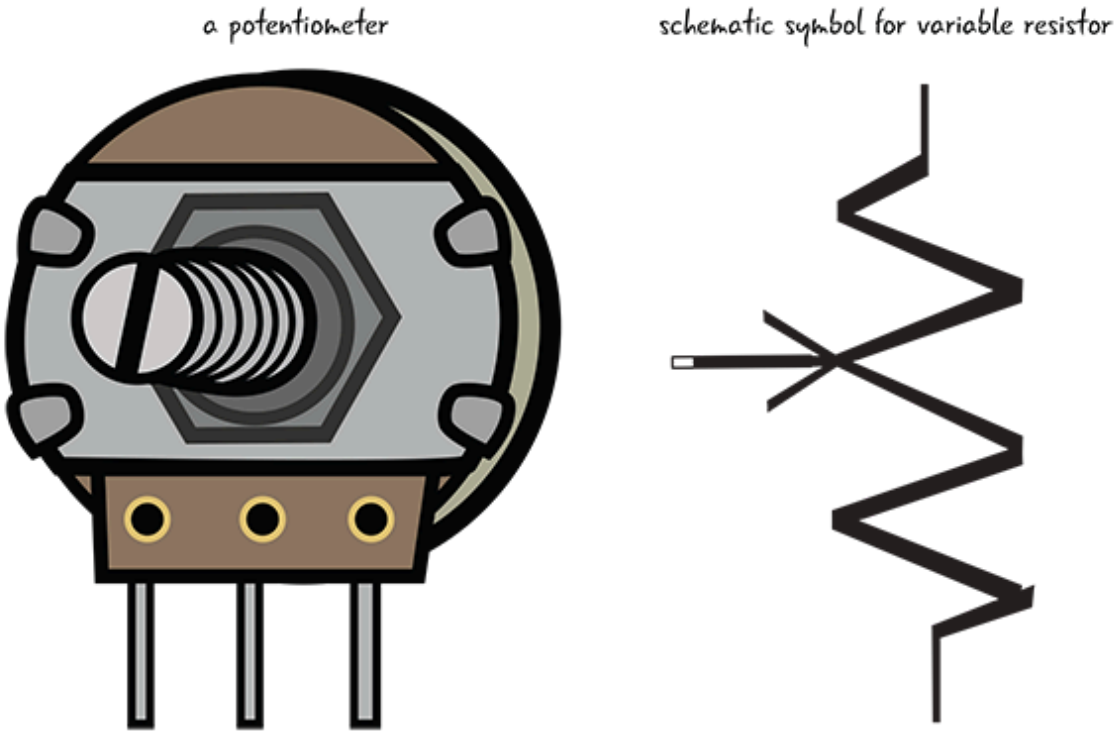


FIGURE 7-4: Component drawing and schematic for the potentiometer

Note

A variable resistor can provide different amounts of resistance.

QUESTIONS?

Q: Is a potentiometer just like the knob on an old TV set?

A: Not exactly. The dial on an old TV has set points, where it stops as you turn it to "tune in" a channel. A potentiometer generally has stop points at both ends, where it has either maximum resistance or minimum resistance. It can be turned smoothly between those two endpoints.

POTENTIOMETER CIRCUIT, STEP BY STEP

The first circuit you will build will contain a potentiometer that controls the brightness of an LED (Figure 7-5).

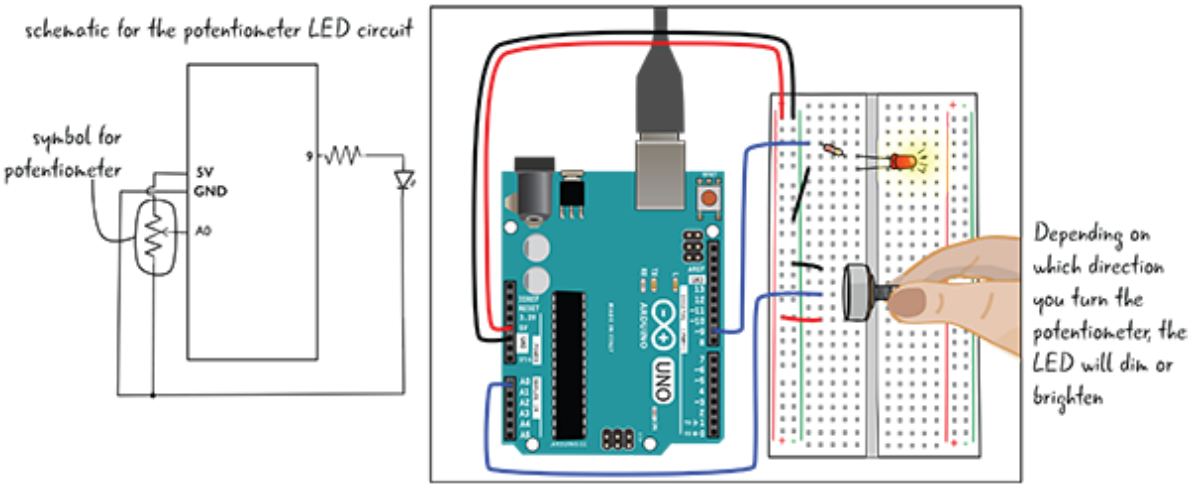


FIGURE 7-5: Completed potentiometer and LED circuit

You'll need these parts:

1 LED

1 220-ohm resistor (red, red, brown, gold)

1 10 kΩ potentiometer

Jumper wires

Breadboard

Arduino Uno

USB A-B cable

Computer with Arduino IDE

You will start with a basic circuit where the anode of an LED is attached through a 220-ohm resistor to the Arduino and the cathode is attached to ground. There is one key difference between this circuit and the circuit you have used in previous chapters: you are using Pin 9 instead of Pin 13 on the Arduino board, as shown in Figure 7-6. We'll explain why soon.

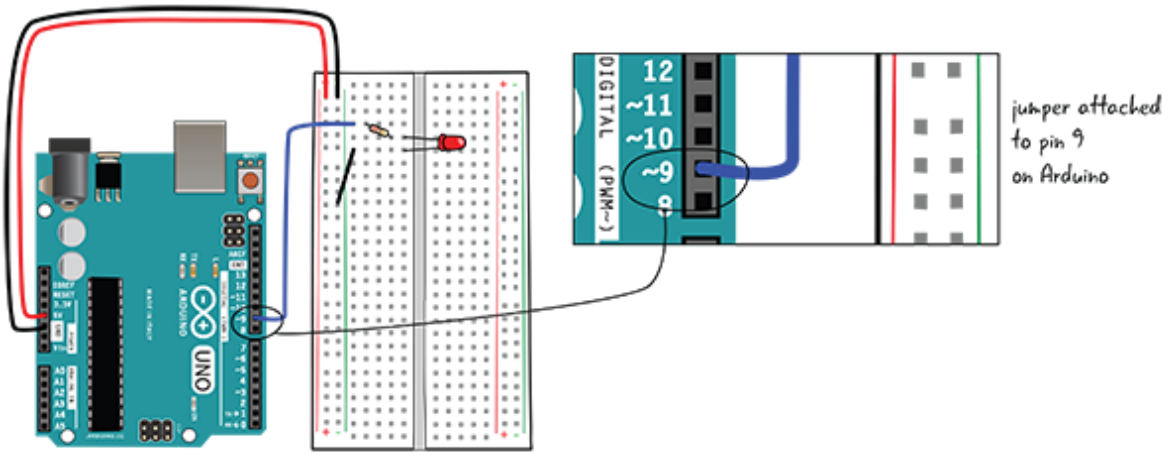


FIGURE 7-6: LED attached to Pin 9

Next you'll place the potentiometer in the breadboard.

ADDING THE POTENTIOMETER

As you have seen, a potentiometer has three pins. In your circuit, you will attach the middle pin to a pin on the Arduino, one of the outer pins to the power bus, and the other outer pin to the ground bus. It doesn't matter which outer pin goes to power and which one to ground.

You will place the potentiometer parallel to the trench, as depicted in Figure 7-7. Each pin of the potentiometer is in a separate row of tie points, with an empty tie point between each of the pins. Orient the potentiometer facing away from the Arduino, with the shaft over the trench, as you can see in Figure 7-7. This will make it easier for you to reach the potentiometer to turn it and to integrate it into the rest of the circuit.

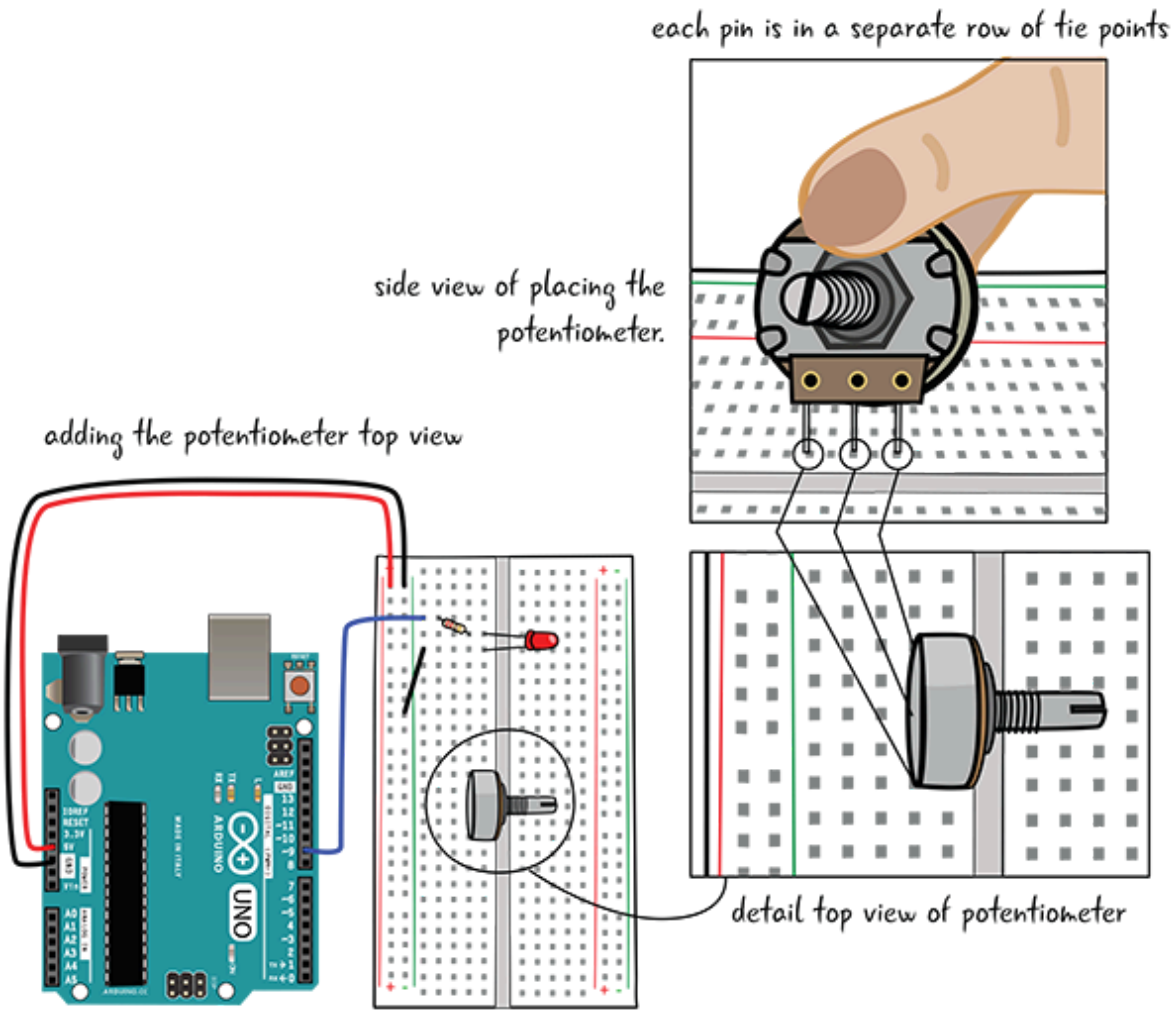


FIGURE 7-7: Attaching the potentiometer

Next you'll attach the potentiometer to the power and ground buses.

Attach the pin at the top of the potentiometer to the ground bus with a jumper. Then, attach the pin at the other edge of the potentiometer to the power bus (Figure 7-8). Make sure the jumpers and the pins of the potentiometer are in the same row of tie points.

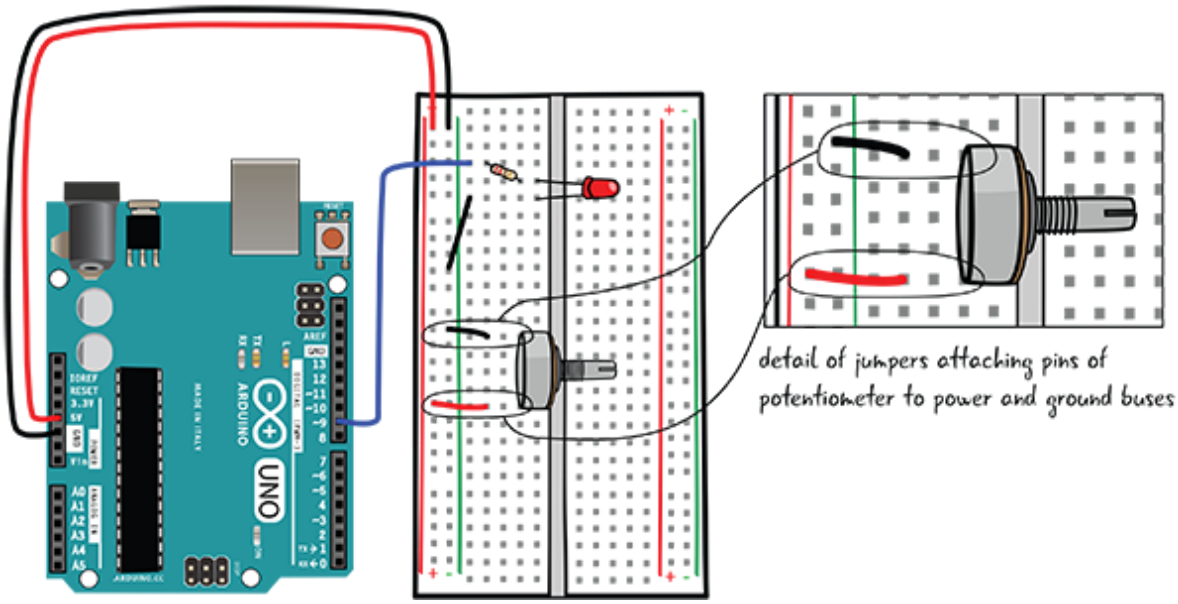


FIGURE 7-8: Adding jumpers to the potentiometer

Finally, attach the middle pin of the potentiometer to Analog Input Pin A0 with a jumper (Figure 7-9).

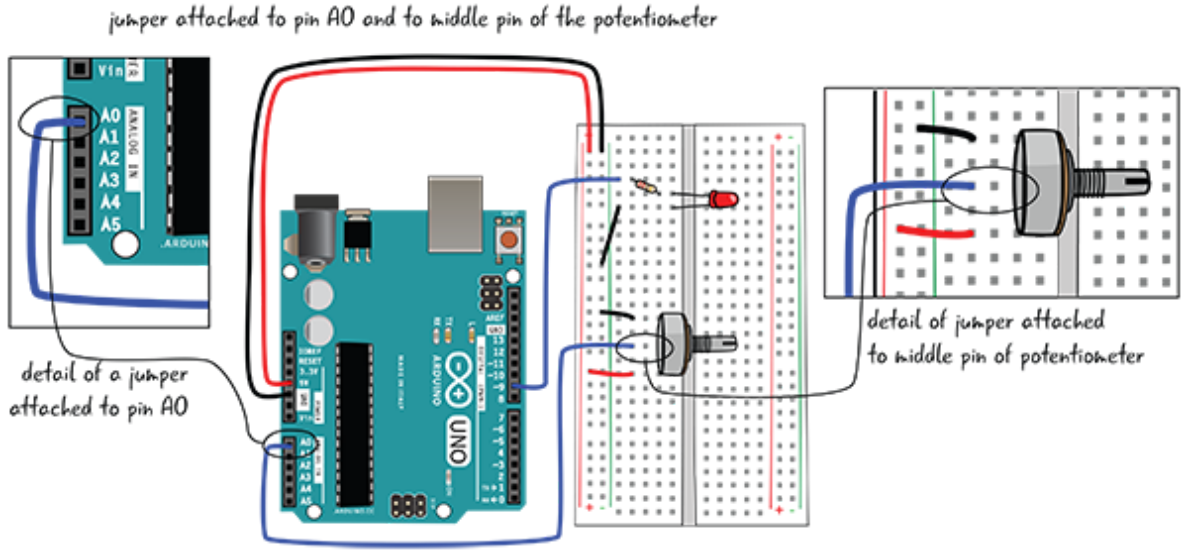


FIGURE 7-9: Attaching the potentiometer to the analog pin

DIM THE LIGHTS

At this point, connect your Arduino to your computer with the USB cable. Let's load up an example sketch from the Arduino IDE. To load the sketch, choose File > Examples > 03.Analog and select AnalogInOutSerial. Save this sketch as LEA7_AnalogInOutSerial.

After you have saved it, click Verify, and then click Upload.

When you turn the potentiometer, the LED should get dimmer or brighter, depending on which way you turn it (Figure 7-10).

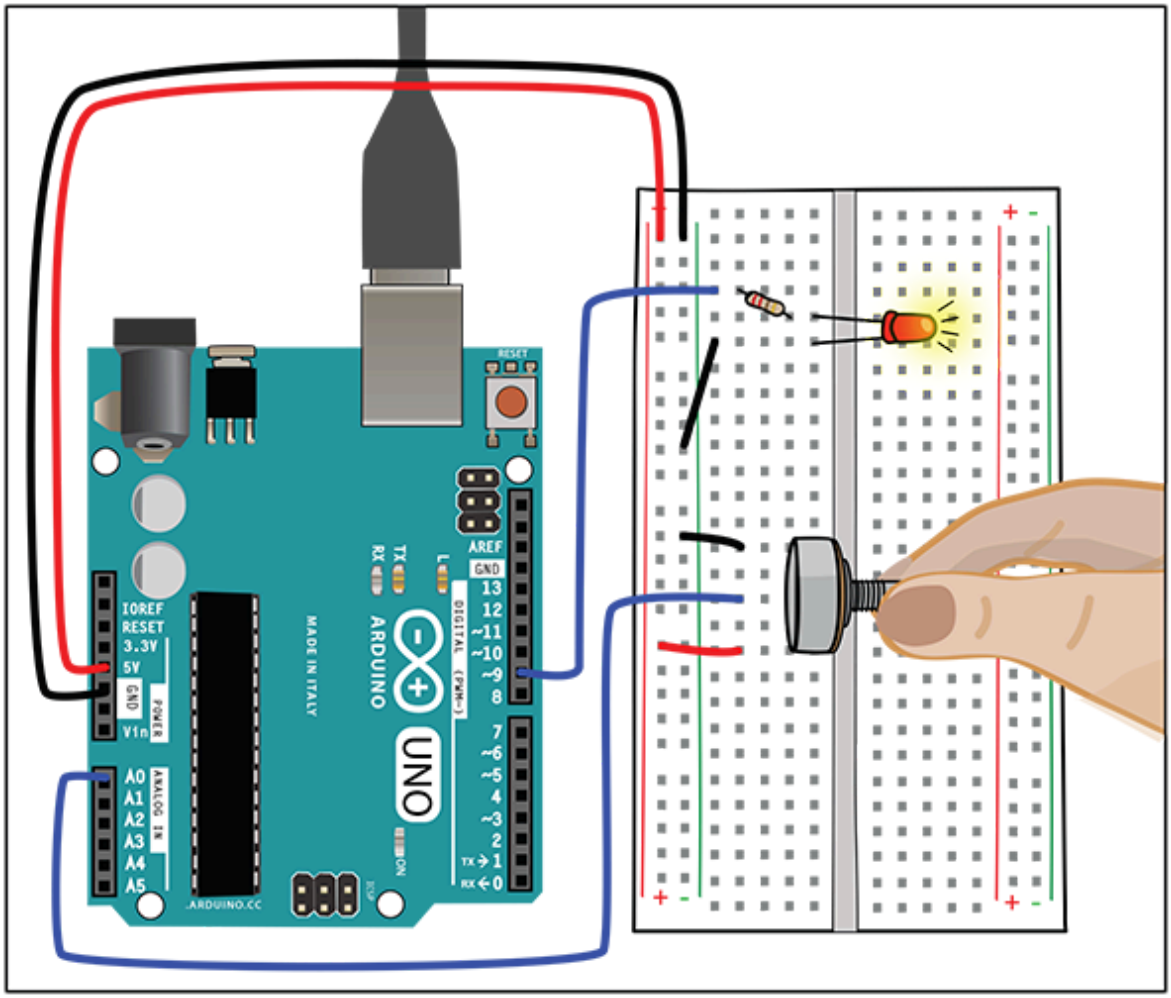


FIGURE 7-10: The LED gets brighter or dimmer when you turn the potentiometer.

THINK ABOUT IT...

You probably use a potentiometer every day for volume control on a stereo or for a dimmer light switch. Can you think of other devices that you might want to try to control with a potentiometer?

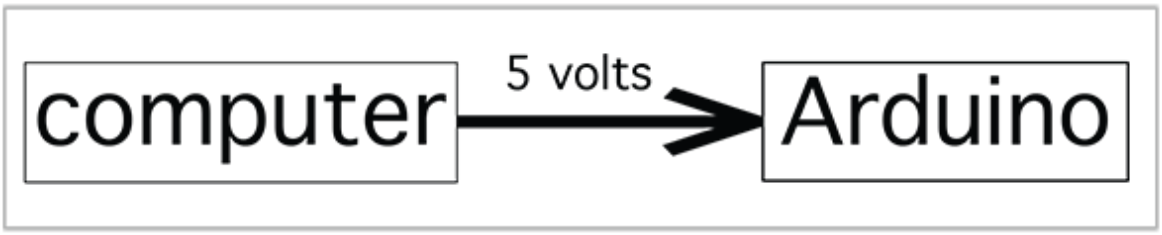
Next you'll see how your sketch allows the circuit to interpret the potentiometer's resistance value and change the LED's brightness accordingly.

WHAT ROLE DOES THE SKETCH PLAY IN YOUR CIRCUIT?

You've seen the circuit in action, so you understand what it does, but how does the Arduino translate the potentiometer resistance value to a brightness value for the LED? To figure that out, let's take a look at how electricity and information flow through the circuit.

Step 1: Power in to the Arduino

Five volts of power come into the Arduino from the computer via the USB cable.

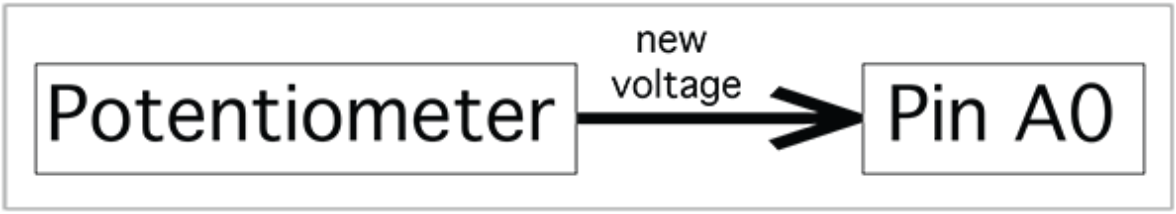


Step 2: Power in to the Potentiometer

Five volts get sent to one side of the potentiometer from the 5v pin on the Arduino (via the power bus).

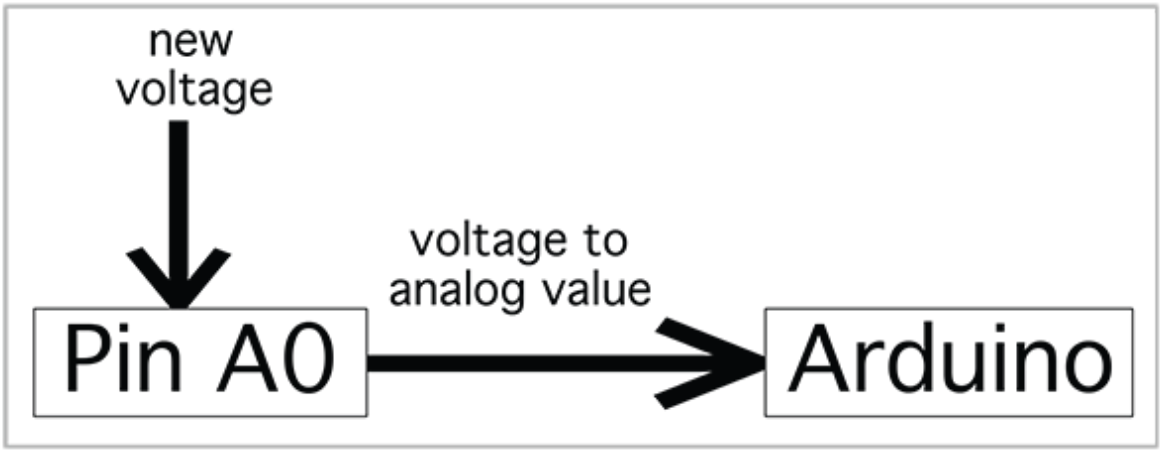
Step 3: Potentiometer Changes the Voltage

The potentiometer creates resistance, lowering the voltage, and then sends this new voltage back to the Arduino via Pin A0.



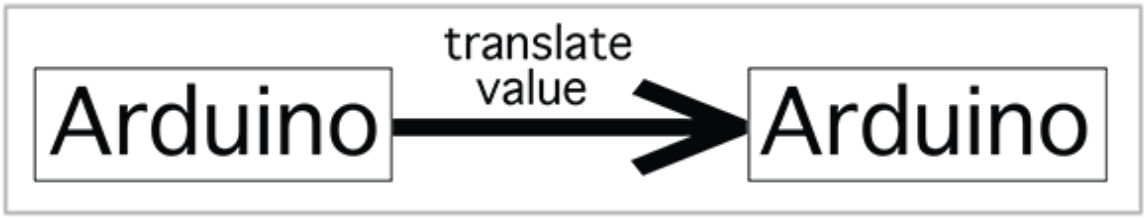
Step 4: Arduino Reads the Voltage

On Pin A0, the Arduino reads the voltage coming in from the potentiometer and translates the voltage value to a number on the 0–1023 analog scale we mentioned earlier. Sometimes this reading takes a longer amount of time. We will talk about how the Arduino determines the value later on in this chapter.



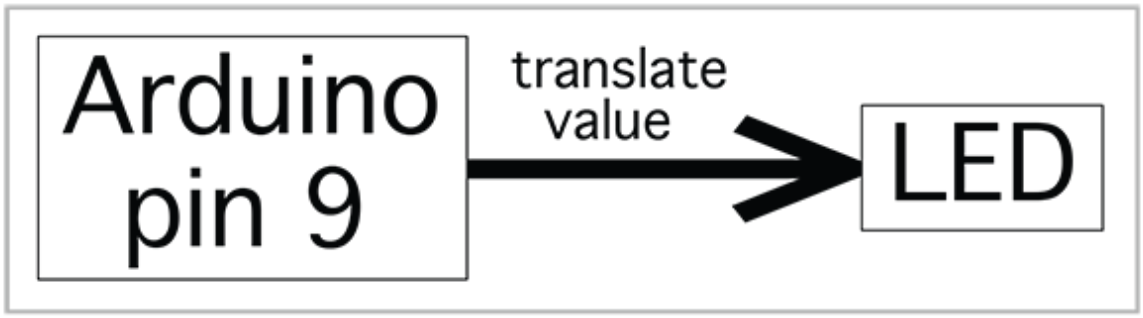
Step 5: Arduino Converts the Value

The Arduino changes the analog value from the potentiometer to a translated analog value using the function `map()`, which we will explain later in the chapter. This step is crucial, since the LED won't understand values between 0 and 1023 but will accept values between 0 and 255.



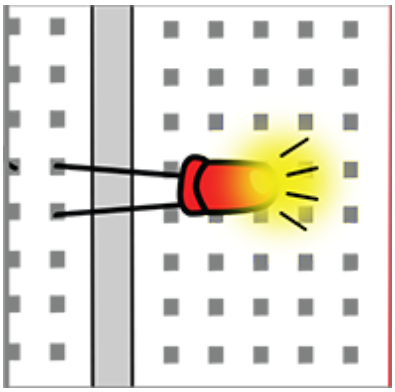
Step 6: Arduino Writes the Value to the LED

The Arduino sends this translated analog value to the LED through Pin 9 using PWM. We will explain what PWM is and how it works later in the chapter.



Step 7: LED Lights Up

The LED lights up; how bright or dim it is will be determined by the analog value it received.

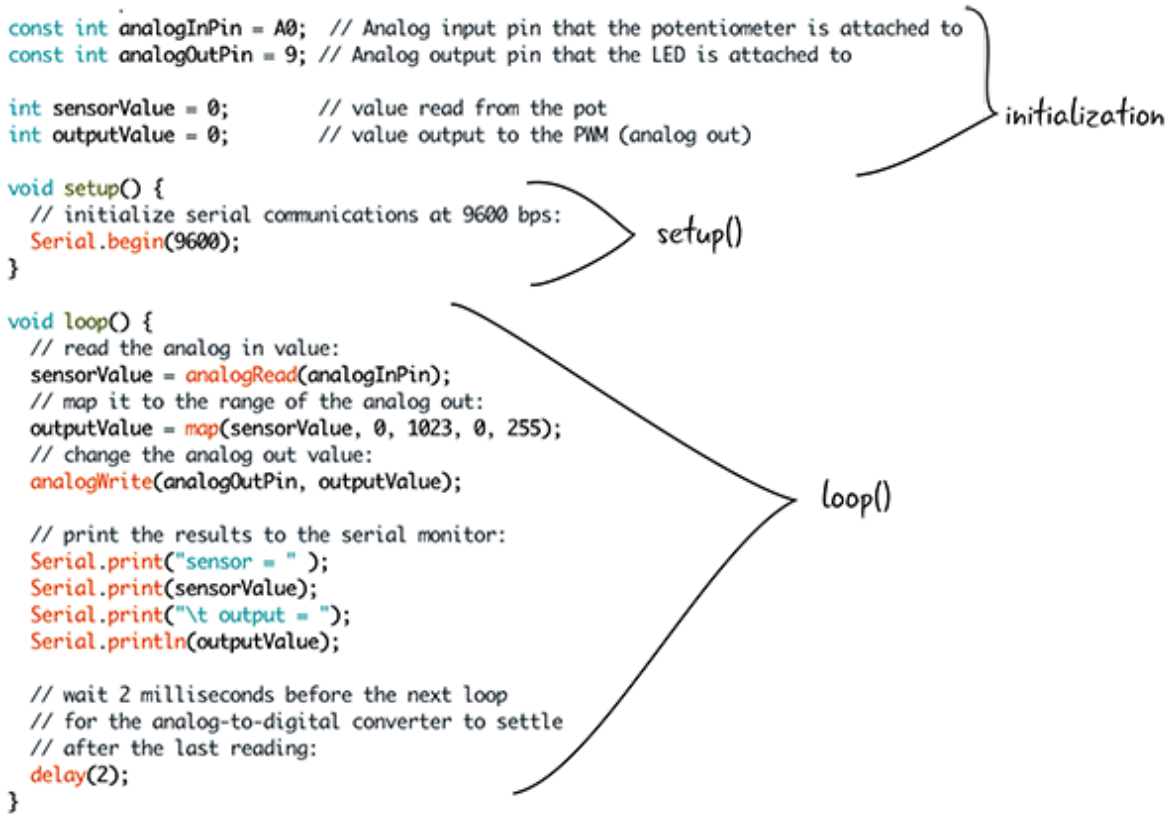


As you can see, the sketch performs the important step of translating the information from the potentiometer into a value that is used to control the brightness of the LED.

Now that you've seen an overview of what's going on in the circuit and how it interacts with the sketch, it's time to dive into the details of the sketch.

THE LEA7_ANALOGINOUTSERIAL SKETCH

This Arduino sketch reads the value of voltage on Pin A0, translates it to a value the LED can understand, and then sends it out to Pin 9. As in other sketches you have seen, there is an initialization section, a `setup()` function, and a `loop()` function. We have again cut out the comments at the top of the sketch.



LEA7_ANALOGINOUTSERIAL INITIALIZATION

The initialization section declares and sets an initial value for some variables you will need in your sketch. As you learned in the previous chapter, when you declare a variable, you give the variable a name, indicate what type of information it will hold, give it a value, and in some cases add a qualifier that indicates whether it is a constant.

```
const int analogInPin = A0; //Analog input pin attached to potentiometer
const int analogOutPin = 9; //Analog output pin attached to LED

int sensorValue = 0;        //value read from the pot
int outputValue = 0;        //value output to the PWM (analog out)
```

As you can see from the excerpted lines of code here, our sketch includes four variables. Here are the details of what each one does:

analogInPin

Holds the pin number that you take the potentiometer reading from. You set this to Pin A0 (Figure 7-11).

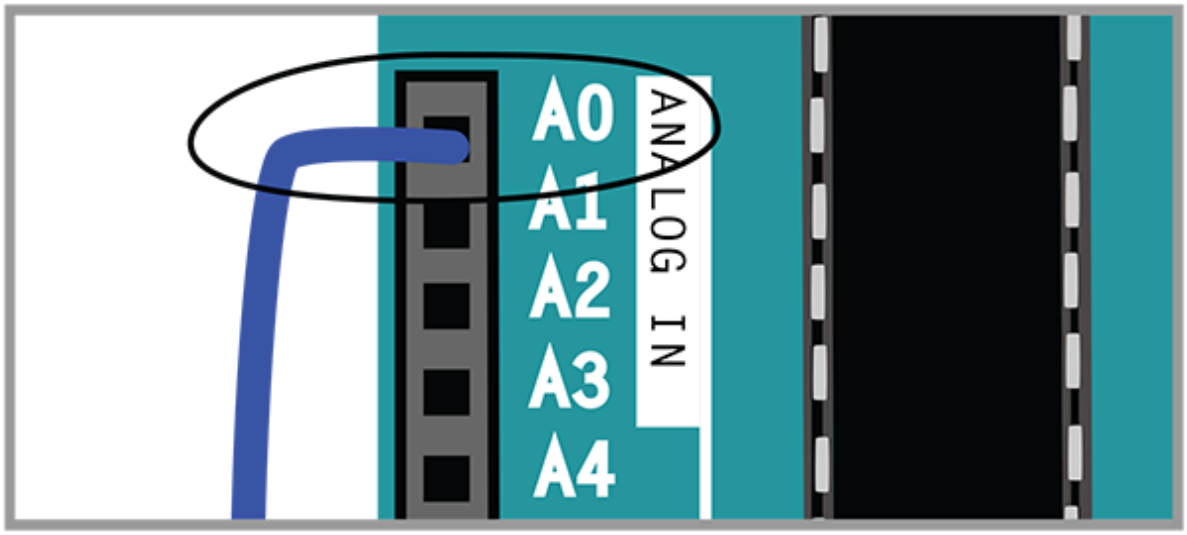


FIGURE 7-11: `analogInPin` set to Pin A0

analogOutPin

Holds the pin number that is connected to your LED. This is set to Pin 9 (Figure 7-12).

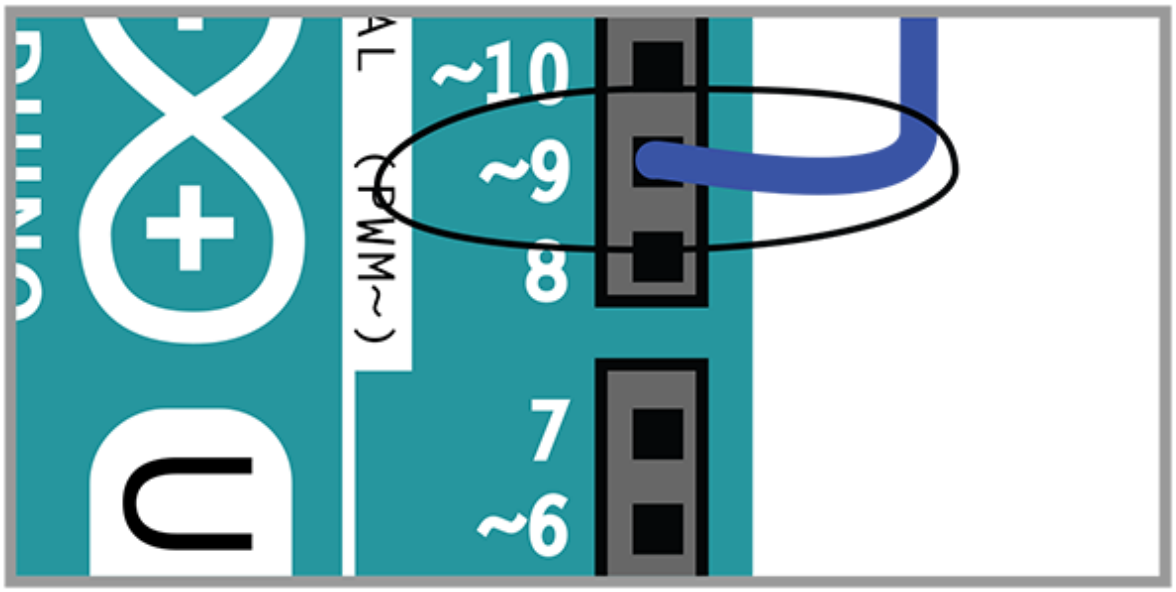


FIGURE 7-12: `analogOutPin` set to Pin 9

sensorValue

This variable is initially set to 0; it will hold the voltage value coming from the potentiometer (Figure 7-13).

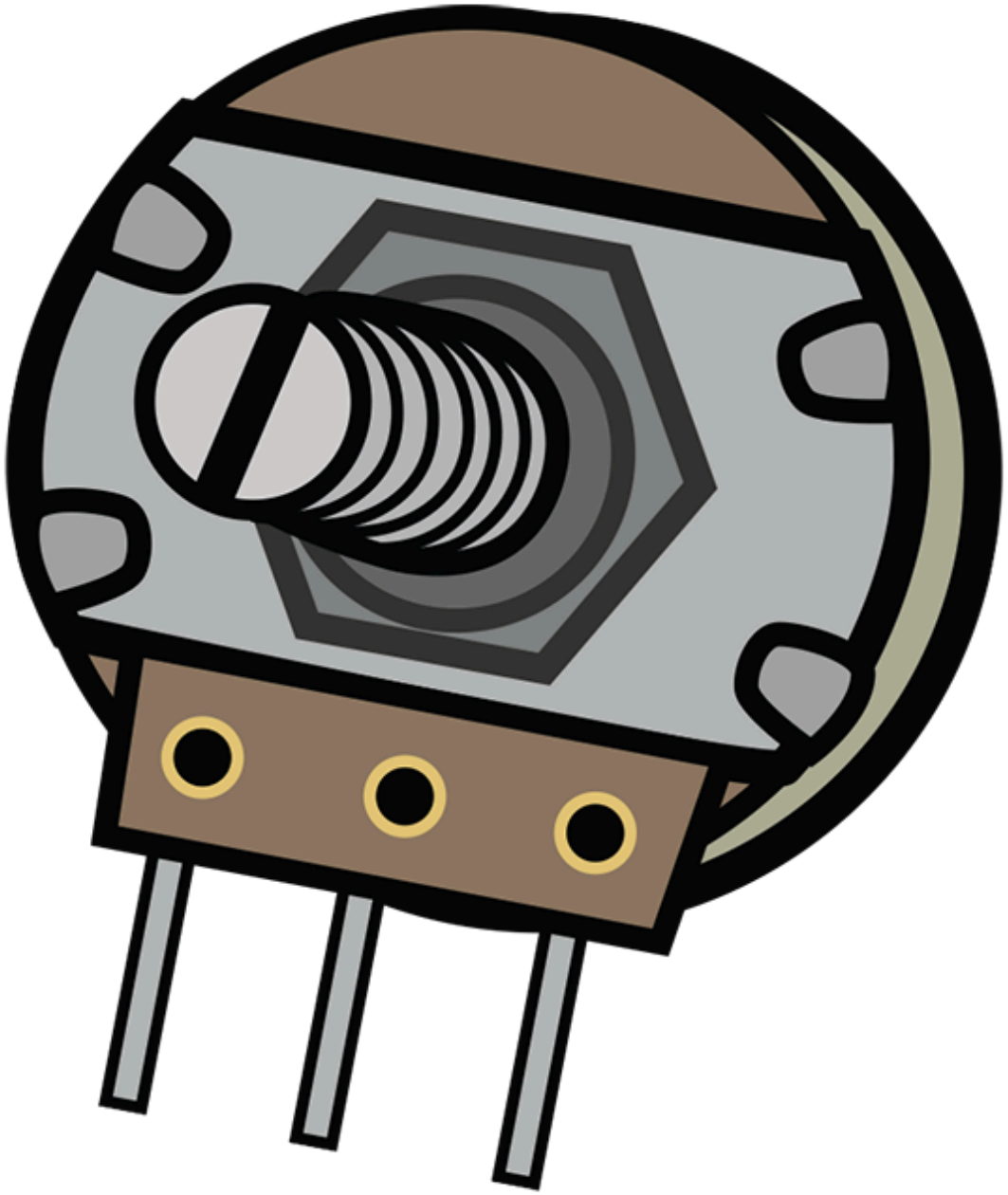


FIGURE 7-13: `sensorValue` will hold the changing voltage level on Pin A0 coming from the potentiometer.

outputValue

This is initially set to 0; it will hold the value the Arduino will be sending to the LED, which determines how brightly it shines (Figure 7-14).

FIGURE 7-14: `outputValue` will hold the value the Arduino will send to Pin 9 to control the brightness of the LED.

The initialization section creates these variables so you can use them later on, in the `loop()` section. Next up: `the setup()` section.

LEA7_ANALOGINOUTSERIAL *SETUP()*

The `setup()` section for the sketch is only one line long, but it is a new Arduino function we have not talked about: `Serial.begin()`. This function uses the serial object.

The serial object is a set of functions and variables that allows the Arduino to communicate with other devices. In this sketch, you will use it to communicate with your computer. `begin()` is a function of the serial object.

> **Note**
>
> A *function* is a way of organizing code or blocks of instructions to the computer.

We talked about functions in Chapter 3 when we discussed `setup()` and `loop()`. Here's how the `begin()` function appears in our `setup()` code:

```
void setup() {
  // initialize serial communications at 9600 bps:
  Serial.begin(9600);
}
```

This line of code tells the Arduino to open a line of communication with your computer (they will communicate through the USB cord that connects them). It also sets a rate of communication for the Arduino and your computer to communicate: 9600 bauds per second (bps). The exact baud rate is not important at this point as long as your Arduino and your computer have a shared rate of communication.

We will look at serial communication more closely in a few pages; for now, let's move on to the `loop()` section of the code.

> **Note**
>
> `begin()` is a function of the serial object that sets up communication between devices.

LEA7_ANALOGINOUTSERIAL *LOOP()* CODE

Here's an overview of what the `loop()` code does:

It takes an analog reading from the pin your potentiometer is connected to and stores it in a variable.

It translates that value into something the LED can understand (a value on the 0–255 scale).

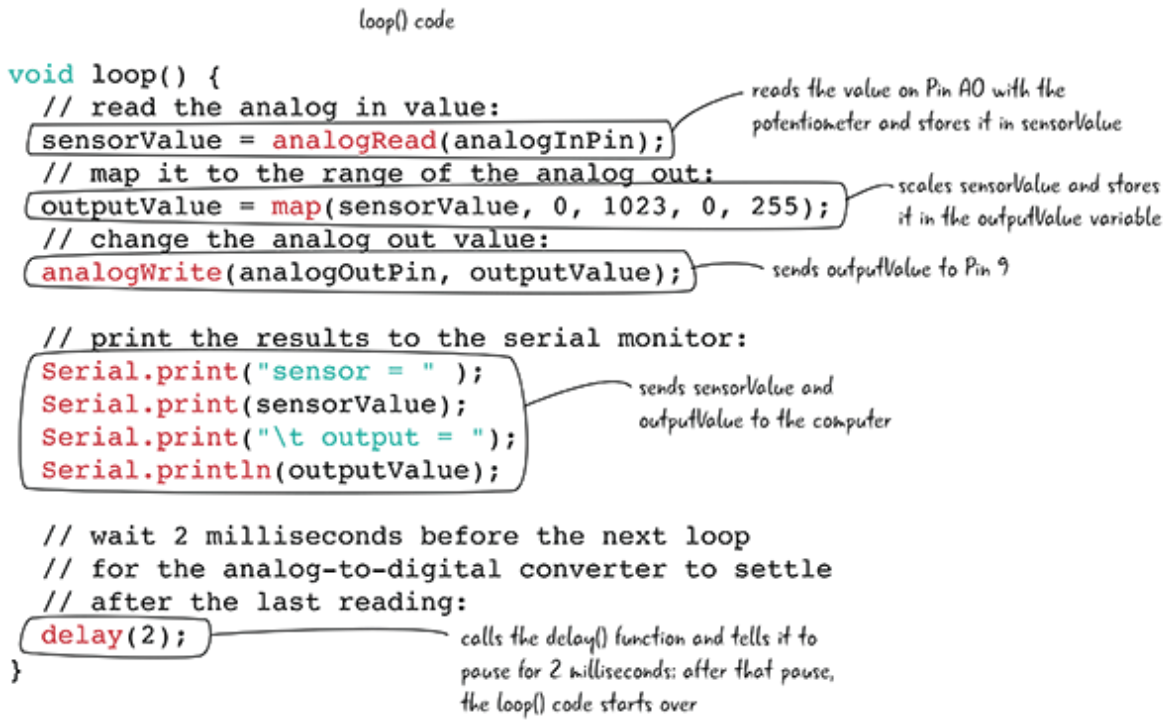
It writes the adjusted value to the LED (Pin 9).

It sends the two values to your computer (`sensorValue` and `outputValue`) so that you can see how they change over time.

It waits a short amount of time (2 milliseconds) before your next reading.

These steps happen in this order repeatedly for as long as your Arduino has power.

Let's look at the code again:



ANALOG INPUT: VALUES FROM THE POTENTIOMETER

The first line of code has the Arduino check the value of voltage coming in from the potentiometer on Pin A0. This value is stored in `sensorValue`.

read value from the potentiometer attached to pin A0

```
// read the analog in value:
sensorValue = analogRead(analogInPin);
```

How is the changing resistance of the potentiometer affecting the values coming out of Pin A0?

If you were to use a multimeter to read the voltage out of Pin A0 when the potentiometer is all the way to one side, giving maximum resistance, you would read a voltage of 0. If the potentiometer were turned all the way to the other side, with no resistance, you would read a voltage of 5 volts. Remember Ohm's law? Here we see it in action, with a changing amount of resistance affecting the amount of voltage.

The values that we get using the analog pins are a *scaled measurement of voltage*. The Arduino converts voltages values between 0V and 5V into a number between 0 and 1023. This process is called *analog-to-digital conversion*.

Figure 7-15 shows a ruler that demonstrates the conversion from voltage to an analog value reading. On the top you see voltage ranging from 0 to 5 volts; on the bottom you see the range you can get from an analog input pin on the Arduino, 0–1023.

Note

Analog input pins read voltage levels from 0V to 5V and convert them to a range of values from 0 to 1023.

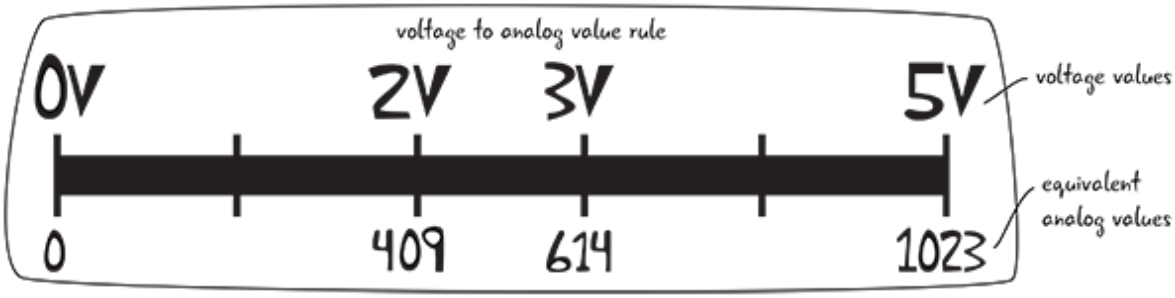


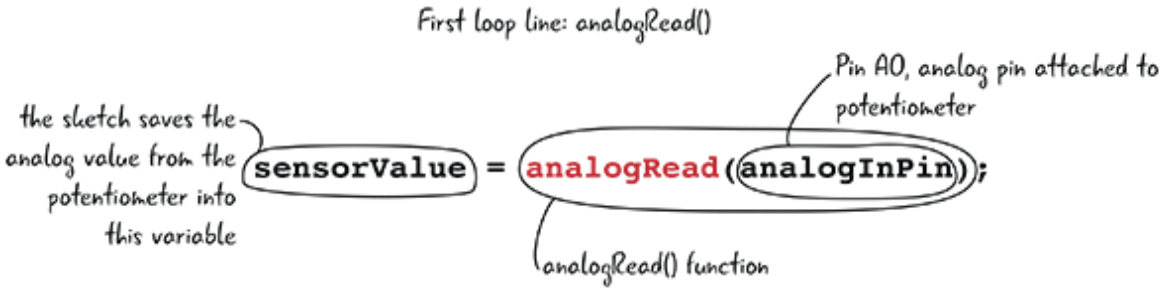
FIGURE 7-15: Converting voltage to an analog reading

On the ruler in Figure 7-15, you can see that at 0 volts you get a reading of 0, and at 5 volts you get a reading of 1023. What happens at the values in between 0 and 5 volts? The analog input value is a number between 0 and 1023, so at 2 volts you get a value of 409. At 3 volts you get a value of 614. These values are automatically calculated by the sketch. You will see how this works with our projects later in this chapter.

Why do you have to convert the value of the voltage? You will see later in this chapter, and in Chapter 8, "Servo Motors," how we use the value (between 0 and 1023) with some of our other functions.

ANALOG INPUT IN CODE: *ANALOGREAD()*

The sketch first takes an analog reading from the pin your potentiometer is wired to, Pin A0, and stores the value of the reading in a variable named `sensorValue`.



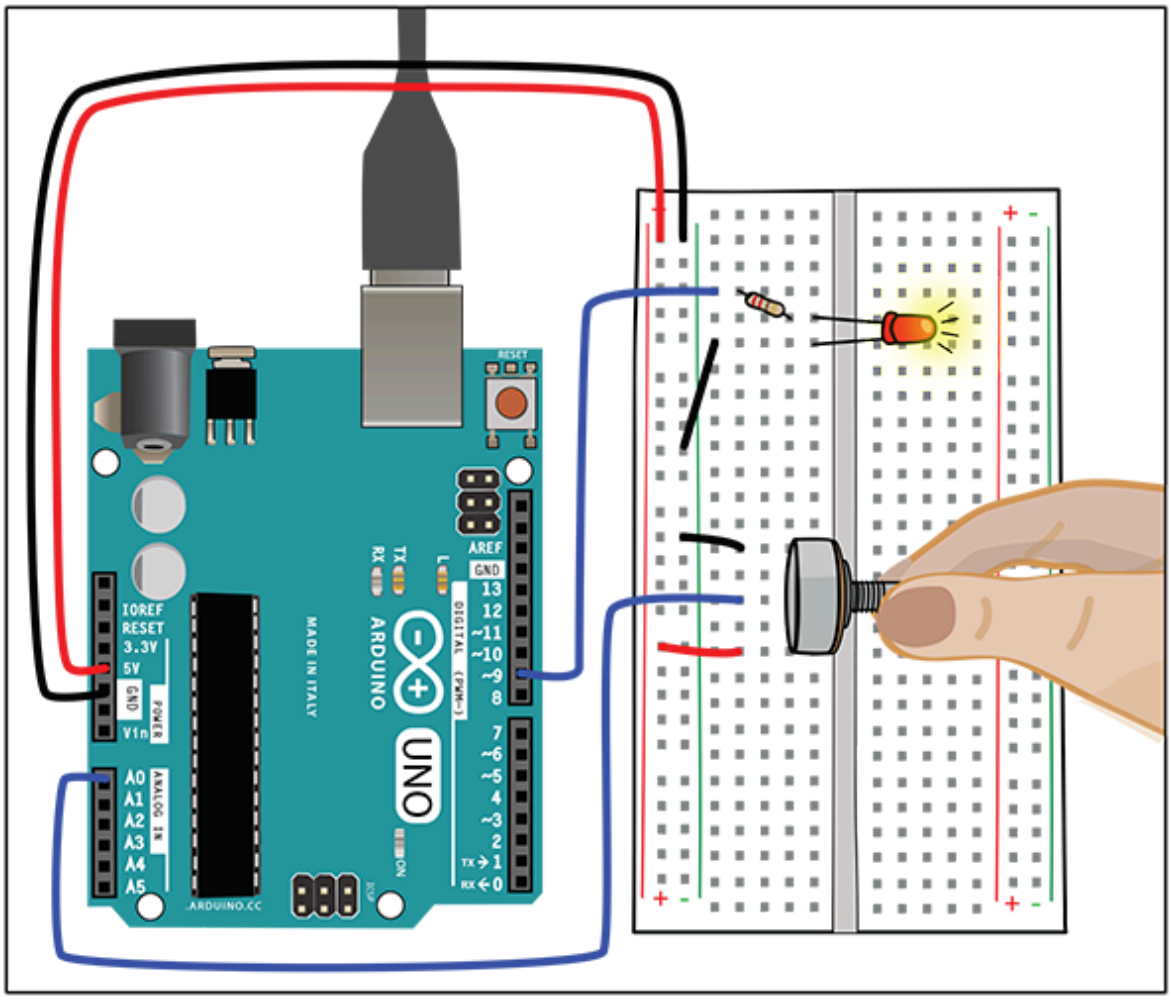


Figure 7-16: Turning the potentiometer fades and brightens the LED.

Remember that analog means that we can have a value other than 0 or 1. `analogRead()` reads a value from an analog input pin that can range between 0 and 1023, as you just learned. If you turn the potentiometer all the way to one side, there is no resistance, so the value is at its highest, which is 1023. When you turn it all the way to the other side, there is maximum resistance, so the value is reduced to 0. The process of reading the value on the pin takes a small amount of time.

Let's bring back our analog value ruler and take a look again (Figure 7-17).

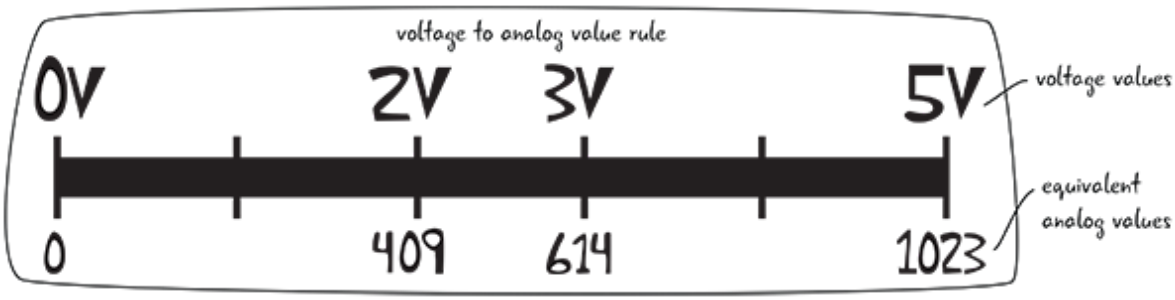


FIGURE 7-17: Voltage-to-analog conversion

If the potentiometer sends 2 volts to the A0, then the value read on that pin is 409. If instead you turn the potentiometer to let 3 volts into the pin, then your new value read will be 614.

Now that you have a better understanding of `analogRead()`, let's look at our next line of code in the sketch.

ADJUSTING VALUES: *MAP()*

The next step is to adjust the value of the `sensorValue` and store it in a new variable. On the next line in our `loop()` function, you see the second variable, `outputValue`, being assigned the value of a `map()` function. The `map()` function automatically scales one sensor value and converts it to another range of values.

loop() code line two: map() function

```
// map it to the range of the analog out:
outputValue = map(sensorValue, 0, 1023, 0, 255);
```

Why are you mapping your value? Why can't you just use the numbers that you get from reading Pin A0? When we looked at the sketch earlier, you saw that you will be using `analogWrite()` to write values to Pin 9. This function takes a range of values from 0 to 255. As the values coming from your analog input pin can range from 0 to 1023, you have to convert that range, 0–1023, to 0–255 in order to pass the value on to Pin 9 (Figure 7-18).

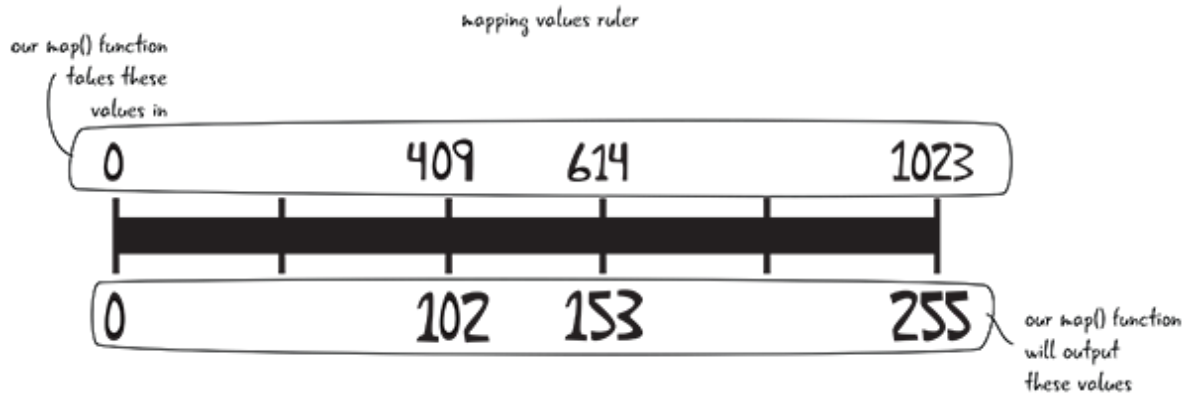
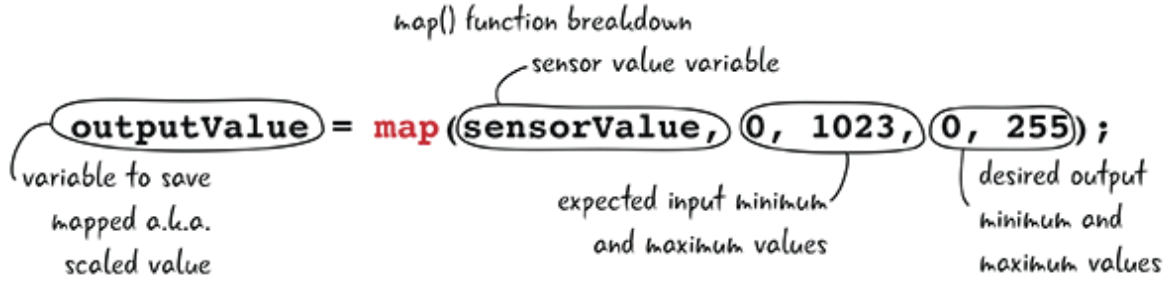


FIGURE 7-18: Converting the range from 0–1023 to 0–255

The `map()` function asks you which variable you want to scale, what the expected minimum and maximum scaled values are for your sensor variable, and what your minimum and maximum values should be.



The value saved to `outputValue` is a scaled-down number. If your sensor reads 1023, `map()` will set the variable `outputValue` to 255. By giving the `map()` function these ranges, almost all the values you save in `outputValue` will be smaller than the original reading indicated. The one exception is if you read 0 on the `sensorValue`—that is still saved in `outputValue` as 0.

> **Note**
>
> The `map()` function is used to scale values from one range to another.

WRITING A VALUE TO A PIN: *ANALOGWRITE()*

Now that you have mapped your value, you're ready to send a value out to your LED and light it up. The next step that the sketch performs is to write the adjusted value to the LED pin.

You'll use the `analogWrite()` function to send analog values to some pins on your Arduino. We'll talk about these special pins in a moment, but first let's take a look at the `analogWrite()` code from our sketch.

loop() code line three: analogWrite()

```
// change the analog out value:
 analogWrite(analogOutPin, outputValue);
```

`analogWrite()` is similar to the `digitalWrite()` function we've already discussed. `analogWrite()` needs to know two things: the pin you want to write to, and the value you want to write to that pin. In this sketch, you are sending the `analogOutPin`, set as Pin 9, a value from the variable `outputValue`, which you set with the `map()` function. By sending an analog value between 0 and 255, the Arduino actually sends a voltage between 0V and 5V back to your LED. Let's look at how the range from 0 to 255 maps to the voltage level on the pin (Figure 7-19).

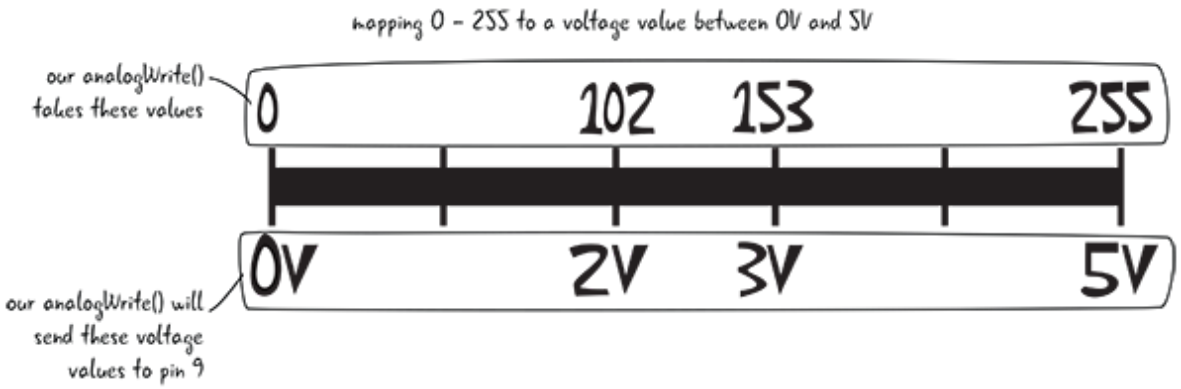


FIGURE 7-19: Mapping 0–255 back to a voltage value

> **Note**
>
> `analogWrite()` takes a value between 0 and 255 and writes a value between 0 and 5 volts to a pin.

The Arduino is able to send an analog value by using a process called PWM. We will explore how PWM works in a few pages.

How Do the Analog and Digital Functions Differ?

Before we get further into `analogWrite()` and how it works with PWM, let's break down how `analogRead()` and `analogWrite()` compare to the `digitalRead()` and `digitalWrite()` functions we have used in previous chapters. We have been making the comparison all along, but Table 7-1 shows how these four functions compare.

TABLE 7-1: Analog and digital functions compared

NAME OF FUNCTION	WHAT IT DOES	ARGUMENTS IT REQUIRES	RANGE OF VALUES
`digitalRead()`	Reads the value of a digital input pin	The number of the pin it is assigned to read	Reads either 1 or 0 from pin
`digitalWrite()`	Writes a value to a digital output pin	The number of the pin it is writing a value to and the value it is writing	Writes either 1 or 0 to pin
`analogRead()`	Reads the value of an analog input pin	The number of the pin it is assigned to read	Reads an integer between 0 and 1023 from pin
`analogWrite()`	Writes a value to an output pin with PWM	The number of the pin it is writing a value to and the value it is writing	Writes an integer between 0 and 255 to pin, which results in a voltage value between 0 and 5 volts

PONDER THIS

Can you think of a circuit you might want to build using analog information? How will it be different from a circuit that uses digital information?

ANALOG VALUES AS OUTPUT: PWM

As you have seen in earlier chapters, the Arduino is capable of putting out only a few different voltage values: either 5V or 3.3V. All of the I/O pins on the Arduino are set to output 5V when used to control circuit components. If the Arduino is capable of producing only 5V on your output pins, how can you create analog values? The Arduino has the built-in capacity to use a technique called *pulse width modulation*, or PWM.

So how does PWM work? Imagine turning the lights on and then off in your room. The room looks bright for a moment, and then dark again. If you continue to flip the light switch back and forth at a slow rate, the room just appears to be bright, then dark, over and over (Figure 7-20).

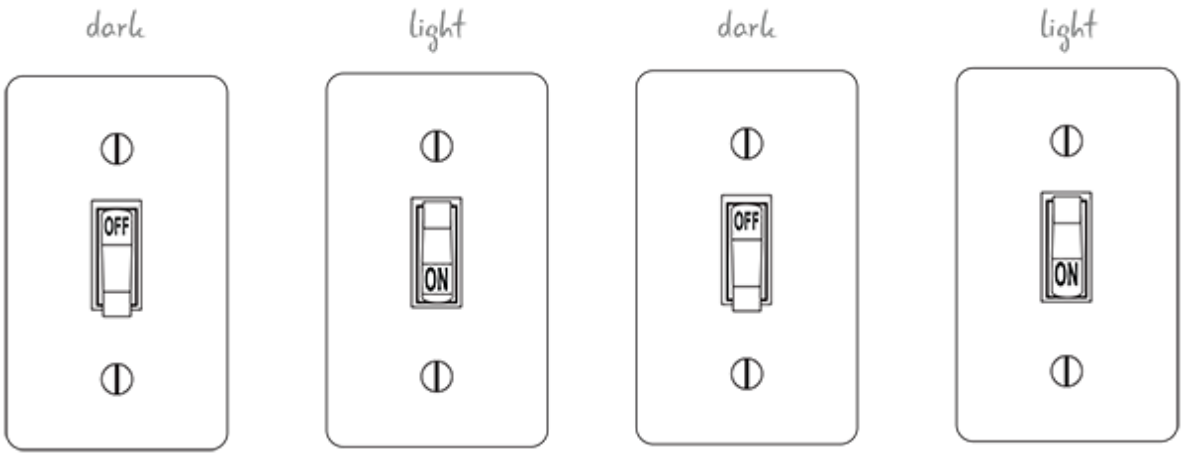


FIGURE 7-20: Flipping light switches

But something strange happens when you flip the light switch faster and faster. Rather than just appearing bright followed by dark, your room will have a light level somewhere in between bright and dark. In fact, the room will also appear brighter if you leave the light on for slightly more time than you leave it off. Your room will have a light level that is the *average brightness* that is dependent on the percentage of time that the light is on in relation to the percentage of time the light is off.

PWM uses a technique similar to flipping the light switch to create a brighter or dimmer light level. When you use PWM on the Arduino, the level of voltage on the PWM pin is switched on and off at various rates at regular intervals. It is sometimes 0 volts, and sometimes 5 volts.

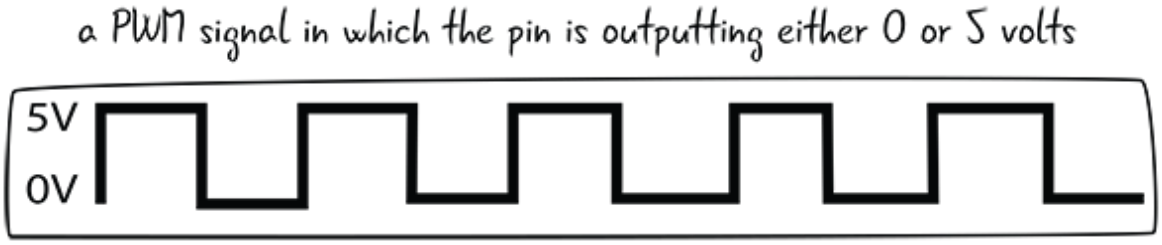


FIGURE 7-21: PWM signal

> **Note**
>
> PWM creates an average value by turning the pin off and on very quickly.

By varying the amount of time the pin is turned on and off, the Arduino creates an average voltage value between 0 and 5.

WHERE ARE THE PWM PINS?

So which pins can you use with PWM? A number of the digital pins on the right side of the Arduino can be used with PWM: Pins 3, 5, 6, 9,

10, and 11. As you can see, each one of the pins is marked by the ~ symbol on the Arduino (Figure 7-22).

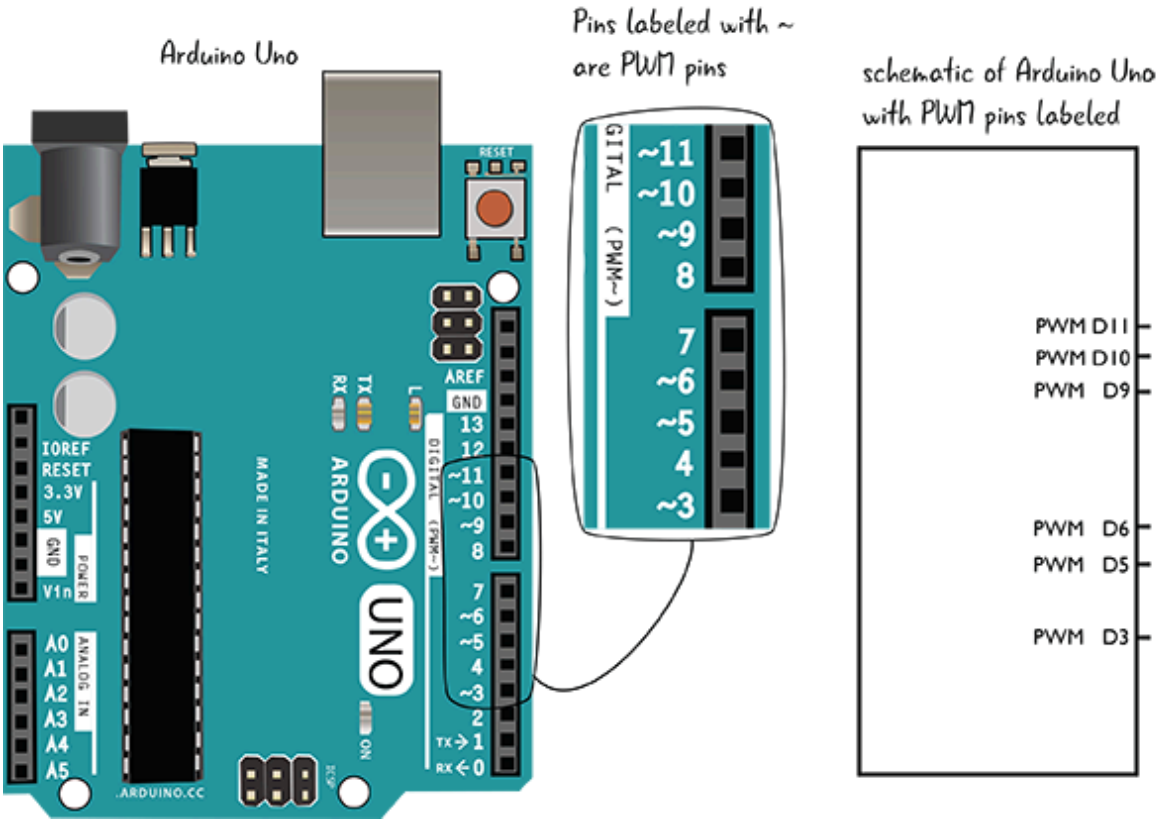


FIGURE 7-22: Labeled PWM pins on the Arduino

QUESTIONS?

Q: Are PWM and `analogWrite()` the same thing?

A: No, PWM and `analogWrite()` are related but they are not the same thing. `analogWrite()` is an Arduino function that tells the Arduino to use a pin to create analog values. This function uses the technique of PWM to create analog values.

Q: PWM turns the pins on and off, and that makes a different value somehow?

A: That's right. Since the Arduino is turning the pin on and off so quickly, the effective value for the voltage that comes out of the pin is the average time the pin is set to `HIGH`. Note that the Arduino is not producing a different value of voltage but rather just using this trick of averages to create the analog value.

Q: The PWM pins can also be used as digital pins?

A: That's right. Depending on your circuit and your sketch, you can use Pins 3, 5, 6, 9, 10, and 11 either as digital pins or as PWM output pins.

SERIAL COMMUNICATION

You've seen how the sketch works with the Arduino to take in information from the potentiometer, change it, and then send it to the pin controlling the LED. Next we'll look at how and why you print values to your computer using serial communication. The second-to-last step the sketch performs is printing two values to your computer (the value from an analog input pin and the value you send to the LED pin) so that you can see how they change over time.

WHY DO YOU NEED TO SEE ARDUINO INPUT AND OUTPUT INFORMATION ON YOUR COMPUTER?

Sometimes it's helpful to find out information about how your sketch is running. This knowledge can be useful if you're trying to debug your circuit. For example, you can see the values you have on your input and output pins. We'll show you how to do this shortly.

WHAT DOES SERIAL MEAN?

Serial, in this context, is a type of communication protocol. It refers to a way that two devices can communicate by sending information across a pair of wires. By changing the value of voltage along a wire from `HIGH` to `LOW`, the Arduino can transmit information across the USB cable to our computer (Figure 7-23). Remember the digital output pins marked TX and RX on the Arduino (digital Pins 1 and 0)? These pins are used to communicate with your computer. TX sends; RX receives.

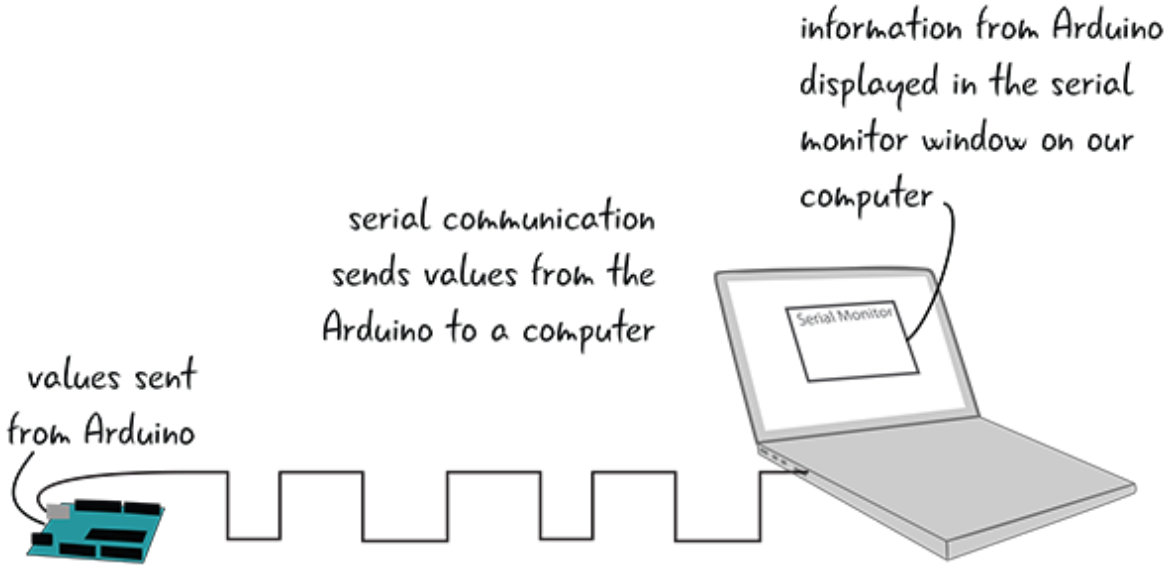


FIGURE 7-23: Serial communication allows the Arduino to talk to a computer.

Serial communication is an easy and efficient method for your Arduino to communicate with your computer. The Arduino IDE

contains a window called the *serial monitor*, which displays the information it receives from the Arduino, such as the values your sensors detect or what function is currently running.

We'll take a look at the serial monitor first before getting back to our code.

USING THE SERIAL MONITOR

The serial monitor is a feature of the Arduino IDE that shows you information sent from the Arduino. It is helpful for debugging and for learning what values a sensor or variable resistor produces. To open the serial monitor, click the button at the top of the Arduino IDE (Figure 7-24).

FIGURE 7-24: Serial monitor button

When you open the serial monitor, you see a window that displays responses from the Arduino, and a drop-down menu that controls the rate of communication, or baud rate, between your computer and the Arduino. As we've mentioned before, the baud rate is the rate of communication that the computer and Arduino use to talk to each other. By default, the baud rate of your serial monitor will be set to 9600, which matches the value you set in the `Serial.begin()` function in your `setup()` code, so you shouldn't have to make any adjustments.

setup()

```
void setup() {
  // initialize serial communications at 9600 bps:
  Serial.begin(9600);
}
```

Note

The Arduino and your computer must use the same rate of communication: the value set in `Serial.begin()`.

Figure 7-25 shows what the serial monitor window looks like when running this sketch.

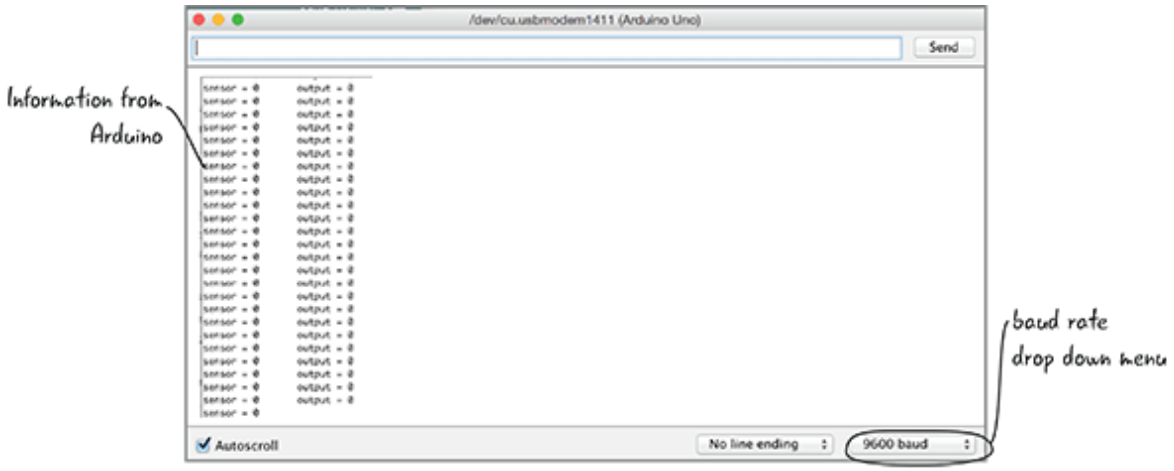


FIGURE 7-25: Running the serial monitor

Now that you understand where to find your serial monitor, let's explore the use of the `serial` object in the `loop()` code.

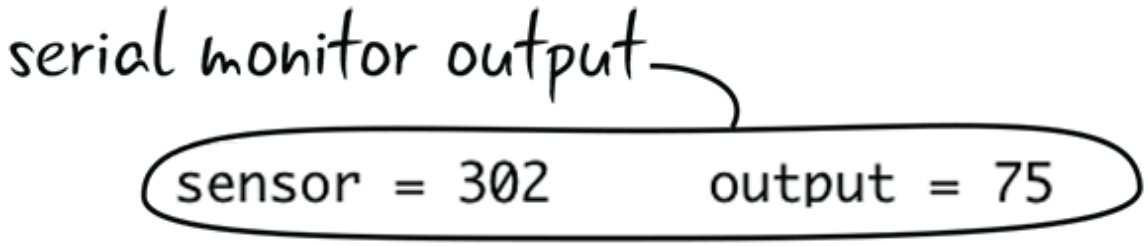
LOOKING AT THE SERIAL CODE

The `serial` object has two functions for sending information to your computer: `Serial.print()` and `Serial.println()`. Our sketch uses both of these functions in order to format the information on the computer screen.

code from loop() for printing to our computer

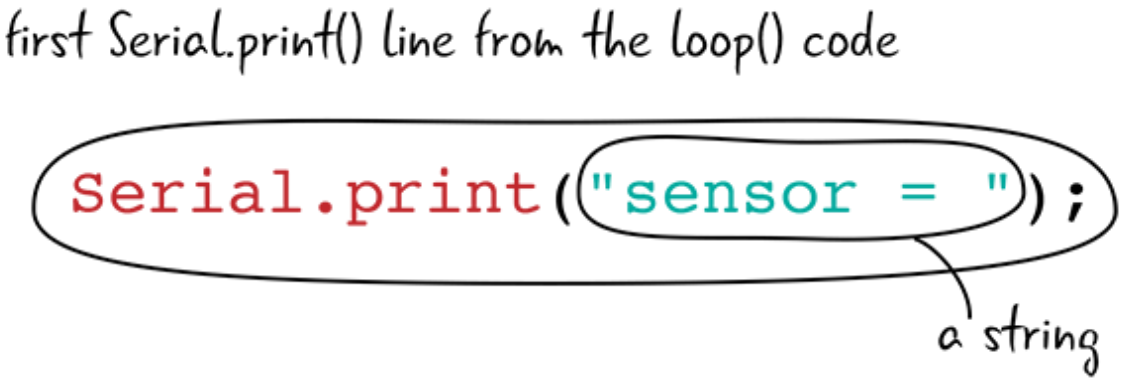
```
Serial.print("sensor = ");
Serial.print(sensorValue);
Serial.print("\t output = ");
Serial.println(outputValue);
```

All four of these lines of code together print the single line in the serial monitor that includes `sensor =`, the value of our sensor, a tab, the text `output =`, and the mapped value. Here's an example of a line of output this code will display in the serial monitor:



Sending Words to the Serial Monitor: Strings

Look at the first `Serial.print()` line, and you see words and quotation marks around them. In order to send words to your serial monitor, you use something called a *string*.



A string is a representation of text in a programming language. Any letters, numbers, or other alphanumeric characters (including spaces and punctuation marks) are represented by strings in your code.

Why do you need strings? Computers normally only work with the value of numbers. Sometimes you need to use text in your code, to pass along textual information or to give context to other data. We will show you how this works as we look more closely at our code in the next couple of pages.

How do you use strings in your code? You place *quotation marks* around a string to identify it. The quotation marks enclose the full group of characters, including all of the letters, spaces, and punctuation.

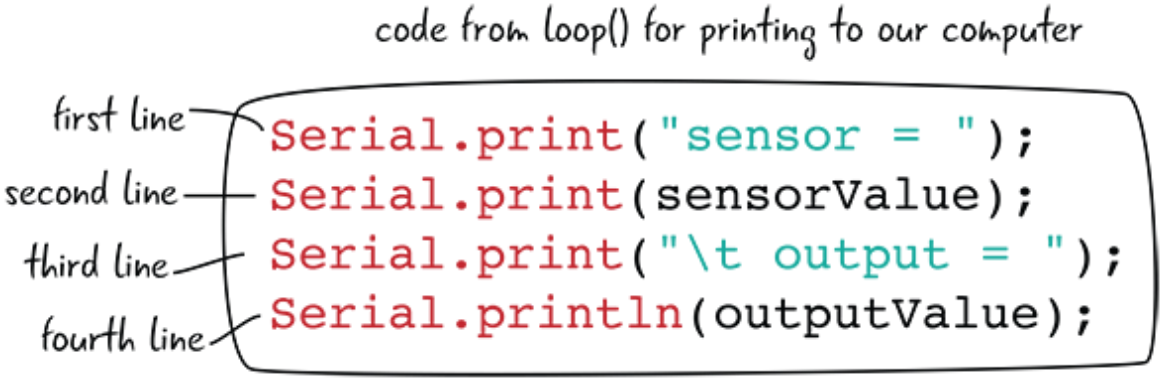
> **Note**
>
> Text is represented by strings in your code. Any alphanumeric characters, including spaces and punctuation marks, are represented by strings.

PRINTING TO THE SERIAL MONITOR

Now we'll take a closer look at how each line of code prints to the serial monitor. You know that any characters inside quotation marks is a string and will be represented as text. You also see your variables

referenced in the code. Let's look at how these work together with `Serial.print()`.

> **Note**
>
> Everything inside the quotation marks, including spaces and punctuation, will be printed to the serial monitor.



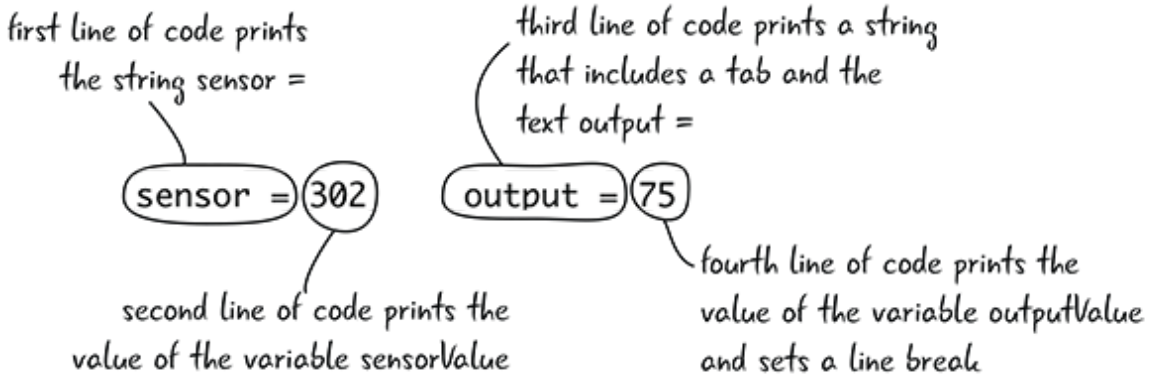
The first line of the code prints the string `"sensor = "` (including the spaces before and after the equals sign).

The second line of `Serial.print()` will print the `value` of the variable `sensorValue`, which is a number. Without quotation marks, the Arduino will print the numeric value stored in the variable, rather than the name of the variable.

The third line of the serial `loop()` code uses quotation marks again, so you know that you are going to print a string. However, you also have a new symbol: what does that "`\t`" mean? The `\t` tells the Arduino serial monitor to include a tab—a set of spaces—in your printed output.

The fourth line prints the value of your `outputValue` variable. But instead of `Serial.print()`, you use `Serial.println()`, which will print a line break.

Remember, the values for `sensorValue` and `outputValue` you see in your serial monitor will change as you turn your potentiometer.



The final line of serial code uses `Serial.println()` rather than `Serial.print()`. You saw that `Serial.println()` automatically adds a line break; how does this affect the way your code appears in the serial monitor?

final Serial.println() line from loop() code

```
Serial.println(outputValue);
```

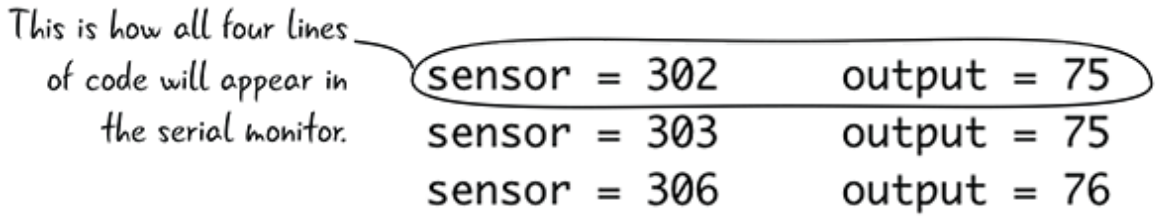
The line break means that the next time you print something to the serial monitor (including the next time through your `loop()` code), the printed serial information will appear on a new line.

The only difference between `Serial.print()` and `Serial.println()` is that line break, which can make it easier to read the information in the serial monitor.

Note

`Serial.println()` **includes a line break, which makes the serial information easy to read.**

And because this code is in `loop()`, the lines will appear over and over again. The values of the variables will change if you turn the potentiometer.



What do those values mean and how do they relate to the scales of numbers we looked at earlier? `sensor` is the value derived from reading the voltage (0–5) on Pin A0 and setting it to a range from 0 to 1023 with the `analogRead()` function. `output` is the value of `sensor` mapped to a range from 0 to 255 by the `map()` function to be used by Pin 9.

THE FINAL *LOOP()* CODE LINE: *DELAY()*

The final step is to wait a short amount of time (2 milliseconds) before your next reading. This is accomplished with a single line of code that includes the `delay()` function you've seen in previous chapters.

2 millisecond delay

```
delay(2);
```

This delay pauses the program for just a moment so that there is enough time to take another sensor reading. There is a limit to how many accurate sensor readings can be taken every second, so the delay helps to space out your sensor readings just long enough to maintain good readings.

> **Note**
>
> A short delay at the end of the sketch keeps your code running smoothly and your sensor readings accurate.

LEA7_ANALOGINOUTSERIAL SUMMARY

As you've seen, the `loop()` code reads an analog value from an analog input pin, scales that value down to a smaller number, writes the analog value out to a PWM pin, and prints the results from all of those steps to your serial monitor so that you can see a readout of how the value changes.

The analog output value—which can be any number between 0V (off) and 5V (fully on)—changes the brightness of your LED. At an

intermediate point such as 3.5V, the LED will be less bright than at 5V. What else can you modulate in this way? Next you'll hook up the speaker and change the notes that come out in a more dynamic way than you did in Chapter 5.

QUESTIONS?

Q: Are there other functions using `Serial` besides `Serial.begin()`, `Serial.print()`, and `Serial.println()`?

A: Yes, there are quite a few, including `Serial.write()` and `Serial.read()`, which are also used to communicate with your computer.

Q: Why do we bother using the special character `\t` to create a tab? Are there other special characters I need to know?

A: Using `\t` makes the output in the serial monitor much easier to read, and that's the only reason we use it. There are many special characters; one that can sometimes be useful is `\n`, which creates a new line. This formats the text in a similar way to using `Serial.println()`—it adds a line break.

Q: I've heard of strings before; they are a way of describing text in other programming languages, right?

A: Yes, alphanumeric characters, including spaces and punctuation, are called strings in many programming languages.

THINK ABOUT IT...

What other information might you want to send from your Arduino to your computer to help you in debugging your projects?

We have sent analog sensor readings in the past, but you can also print strings (to check if something happens, like a button press) or digital readings.

You've seen how your potentiometer can give you a range of values when hooked up to an analog input, and you know how to map those values in your sketch to get values you can use with a PWM output pin. Now let's add a speaker to your circuit to control tones with analog values.

ADDING THE SPEAKER

You're going to keep all the components that are currently in your circuit and add a speaker (Figure 7-26). Both the LED and the speaker will use the values produced by turning the potentiometer to control their behavior.

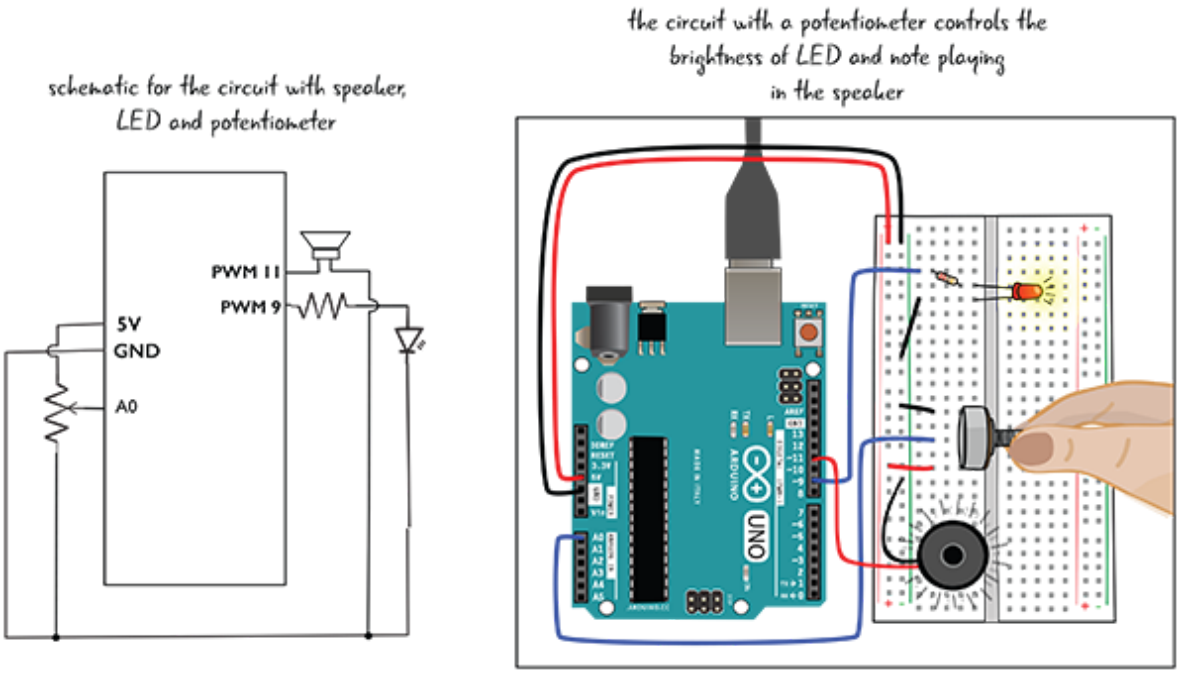


FIGURE 7-26: Adding the speaker to our circuit

Part to add

1 8-ohm speaker

Connect one end of the speaker to Pin 11 and the other to ground. Remember, the speaker doesn't have any orientation (Figure 7-27).

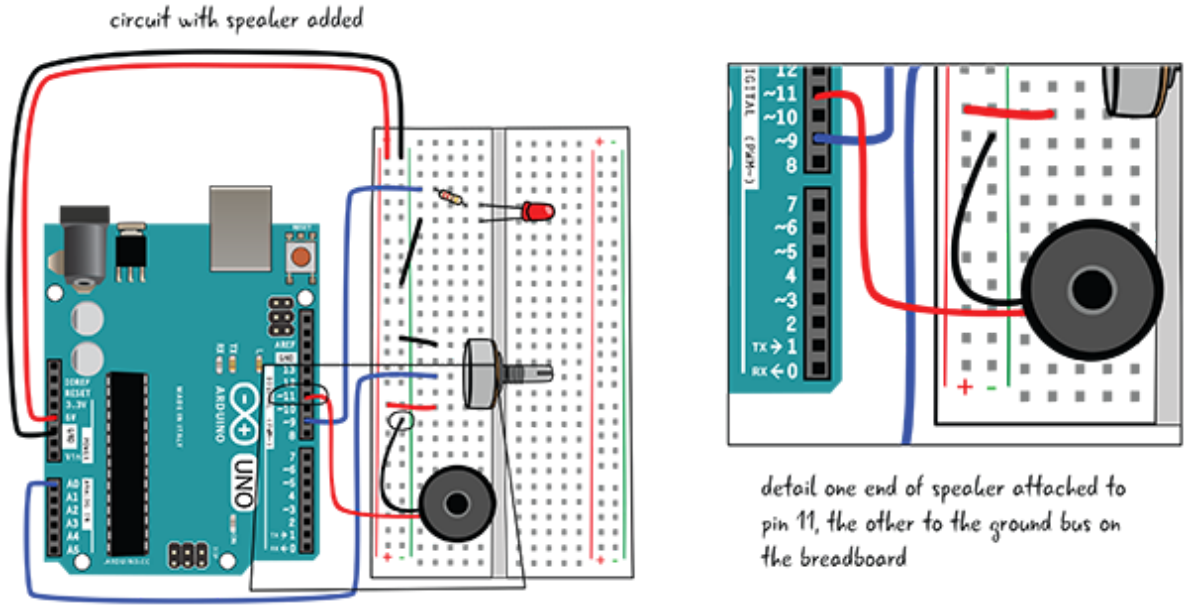


FIGURE 7-27: Attaching the speaker to the potentiometer circuit

Once you've added the speaker to the circuit, attach your computer to the Arduino and open up the LEA7_AnalogInOutSerial sketch. You will be adjusting it.

UPDATING YOUR CODE

Save your sketch as LEA7_VariableResistorTone. You have to add two lines of code to use your speaker: in the initialization section, a variable to hold the value of the pin attached to the speaker, and in the `loop()` section, a call to the `tone()` function. You'll also comment each line to explain what it does.

```
// Analog input pin that the potentiometer is attached to
const int analogInPin = A0;
// Analog output pin that the LED is attached to
const int analogOutPin = 9;
// Analog output pin that the speaker is attached to
const int speakerOutPin = 11;

int sensorValue = 0;      // value read from the pot
int outputValue = 0;      // value output to the PWM (analog out)

void setup() {
  // initialize serial communications at 9600 bps:
  Serial.begin(9600);
}
void loop() {
  // read the analog in value:
  sensorValue = analogRead(analogInPin);
  // map it to the range of the analog out:
  outputValue = map(sensorValue, 0, 1024, 0, 255);
  // change the analog out value:
  analogWrite(analogOutPin, outputValue);
  //call to the tone function
  tone(speakerOutPin, sensorValue);
  // print the results to the serial monitor:
  Serial.print("sensor = " );
  Serial.print(sensorValue);
  Serial.print("\t output = ");
  Serial.println(outputValue);

  delay(2);
}
```

variable to hold the pin attached to the speaker

calls the tone() function

LEA7_VARIABLERESISTORTONE

Once you have added those lines of code (initialized the variable to hold the speaker pin, added a call to the `tone()` function, and commented each line), attach your computer to the Arduino. Verify and then upload your sketch.

Note again that you are using the potentiometer to set the pitch of the audio coming out of your speaker (Figure 7-28). As you turn it, the pitch changes, getting higher as the LED gets brighter and lower as the LED dims.

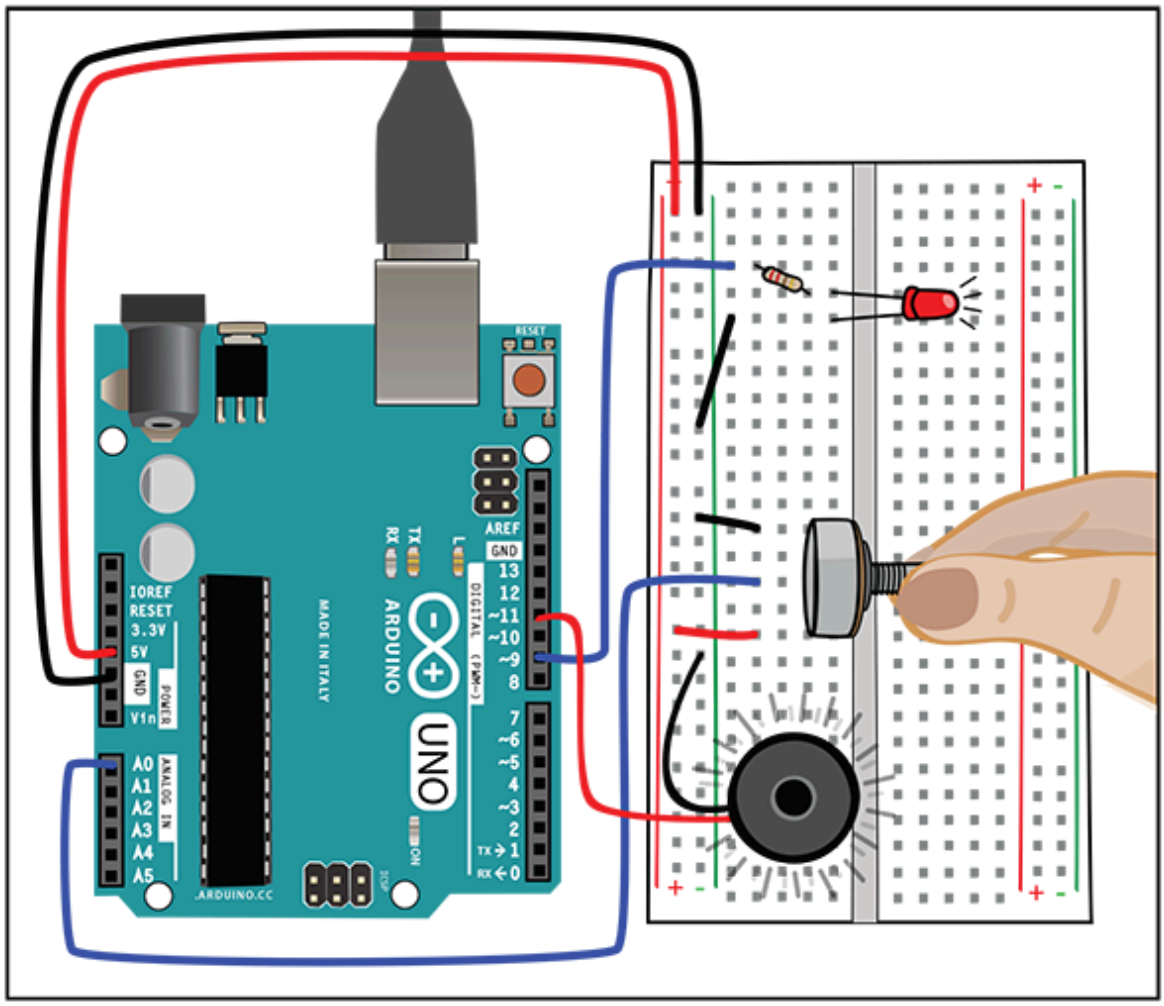


FIGURE 7-28: Turning the potentiometer changes the pitch.

Before we move on to replacing the potentiometer with a photoresistor to build the theremin, let's take a closer look at the call to the `tone()` function.

```
tone(speakerOutPin, sensorValue);
```

You may remember from Chapter 6 ("Switches, LEDs, and More") when you used the `tone()` function that it takes two arguments: the pin the speaker is attached to—in this case, the variable

`speakerOutPin` (set to Pin 11)—and the frequency of the tone to be played, here set to the variable `sensorValue`, which is the value derived from reading the potentiometer on Pin A0. You don't need to map this value to the smaller scale, since the range of frequencies accepted by the `tone()` function is much wider than 0–255.

Now that you've built your circuit with the potentiometer and the speaker, you will swap out the potentiometer for a photoresistor to create your theremin.

ADDING THE PHOTORESISTOR

Place the photoresistor in the breadboard so that one end is in the same row of tie points as the jumper from Analog Pin A0 (Figure 7-29). The other end should be in the row of tie points below that. The photoresistor doesn't have an orientation, so don't worry about placing it in the breadboard backward.

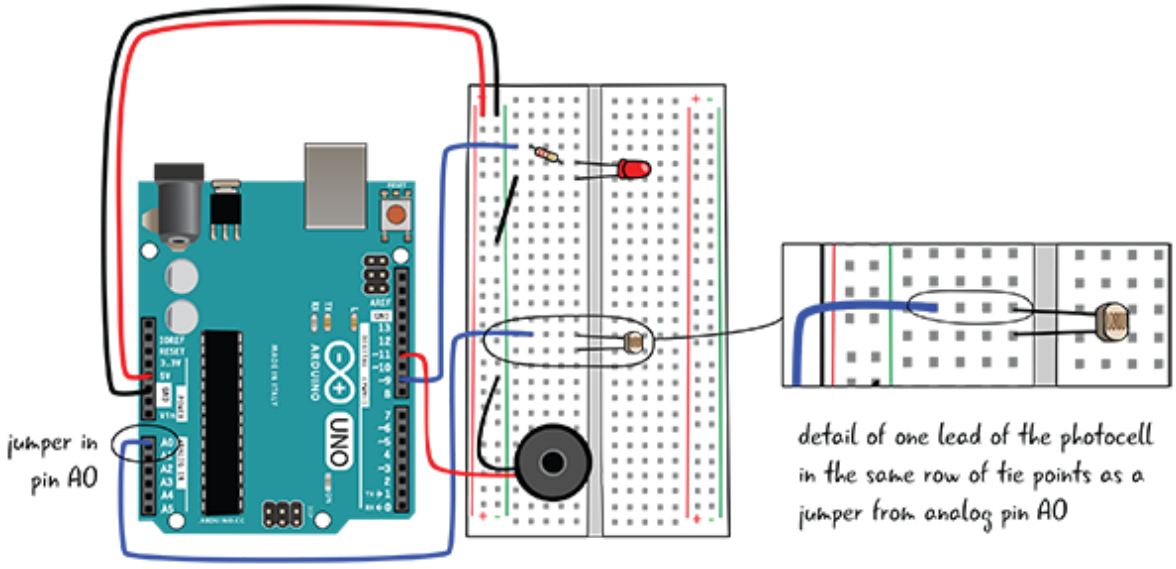


FIGURE 7-29: Adding the photoresistor to the circuit

Note

Photoresistors don't have an orientation and can't be placed backward in your circuit.

Next, add a jumper that connects the other end of the photoresistor to the power bus (Figure 7-30).

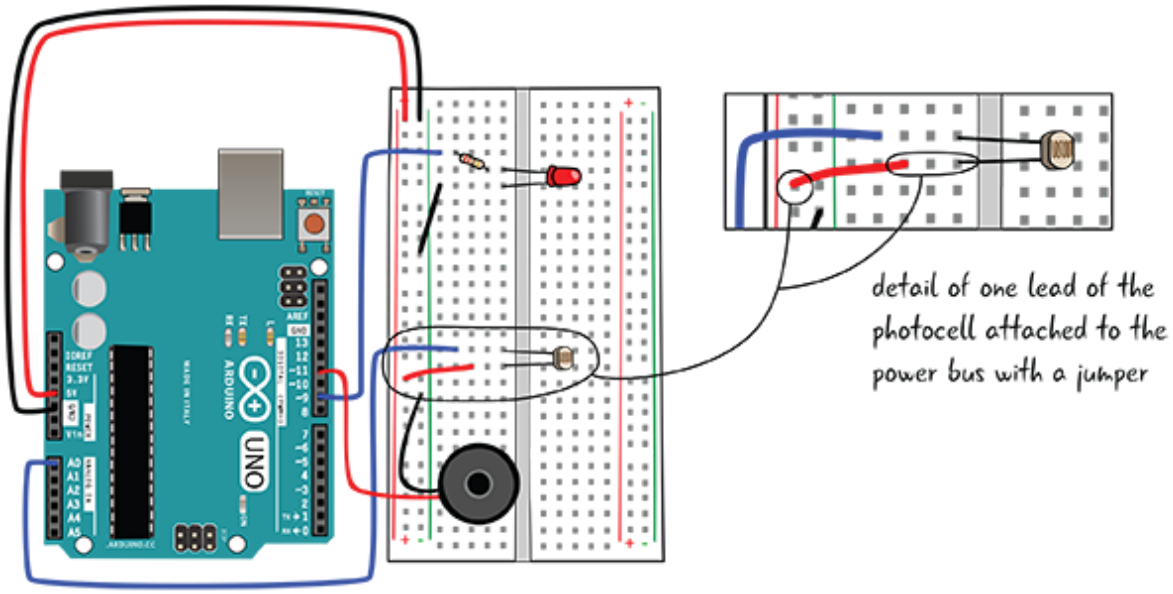


FIGURE 7-30: Adding the jumper wire to power

Now add the 10 kΩ resistor to the same row of tie points as the jumper to Pin A0 and one end of the photoresistor (Figure 7-31). The other end of the 10 kΩ resistor is jumped to the ground bus.

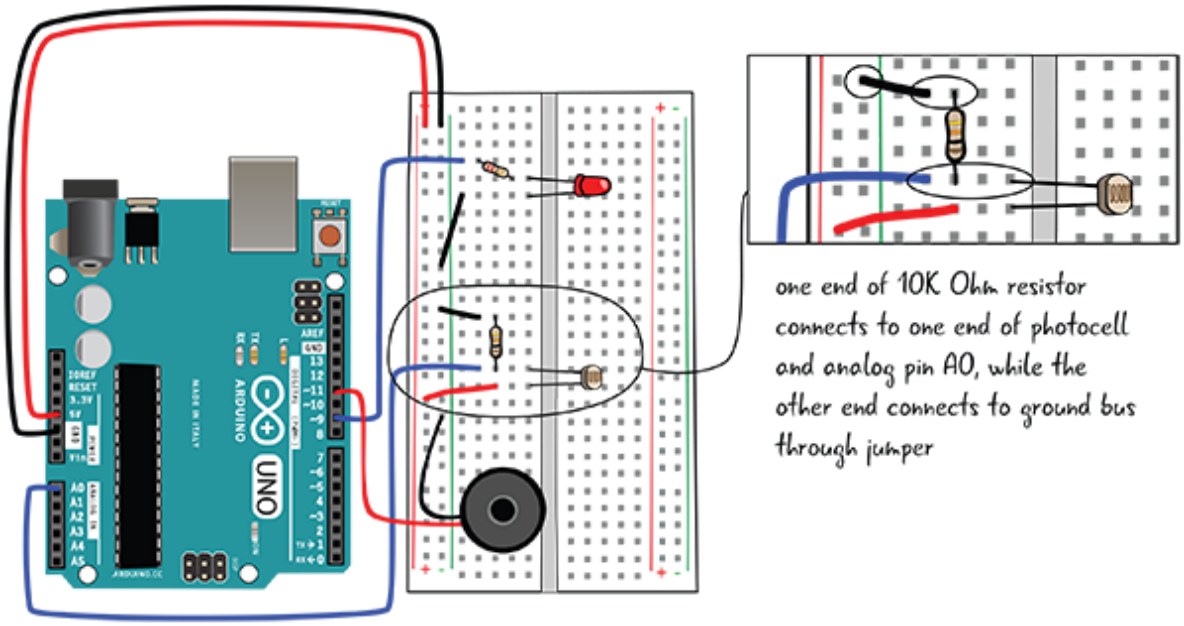


FIGURE 7-31: 10 kΩ resistor added to circuit

You have now completed the circuit. Attach your computer to the Arduino through the USB cable and see what happens (Figure 7-32).

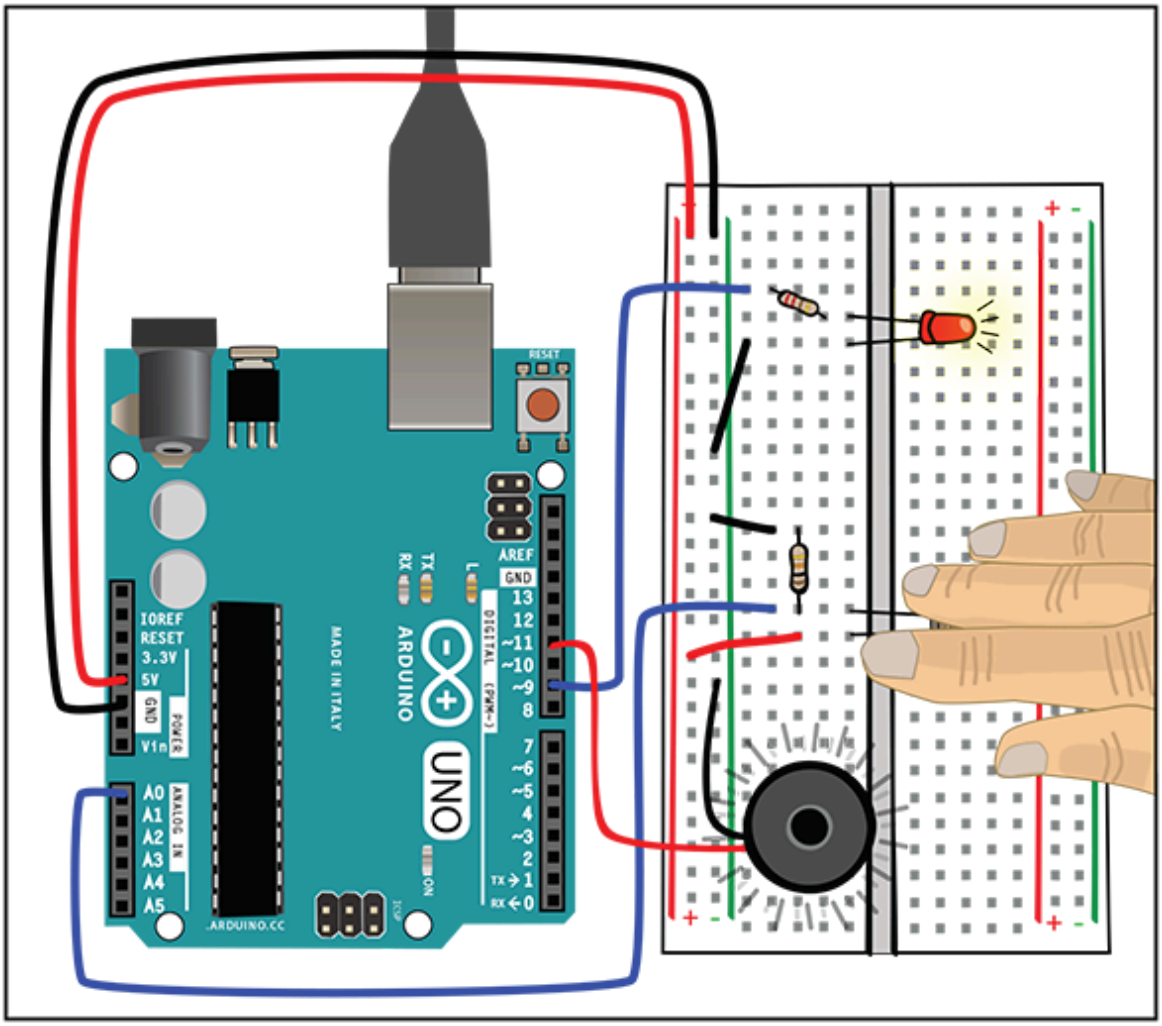


FIGURE 7-32: Testing the completed circuit

VOLTAGE DIVIDER

The arrangement of the photoresistor with the resistor in series is an example of a very common circuit called a *voltage divider*. Voltage dividers are helpful when using some sensors, like the photoresistor, but they won't be required for all circuits. To understand how the voltage divider functions, think of it as changing a large voltage into a smaller one.

Why didn't you need another resistor when you used the potentiometer in your circuit? It contains a resistor with a wiper, which divides the resistor in half (see Figure 7-33). Moving the wiper adjusts the ratio of resistance of the two halves.

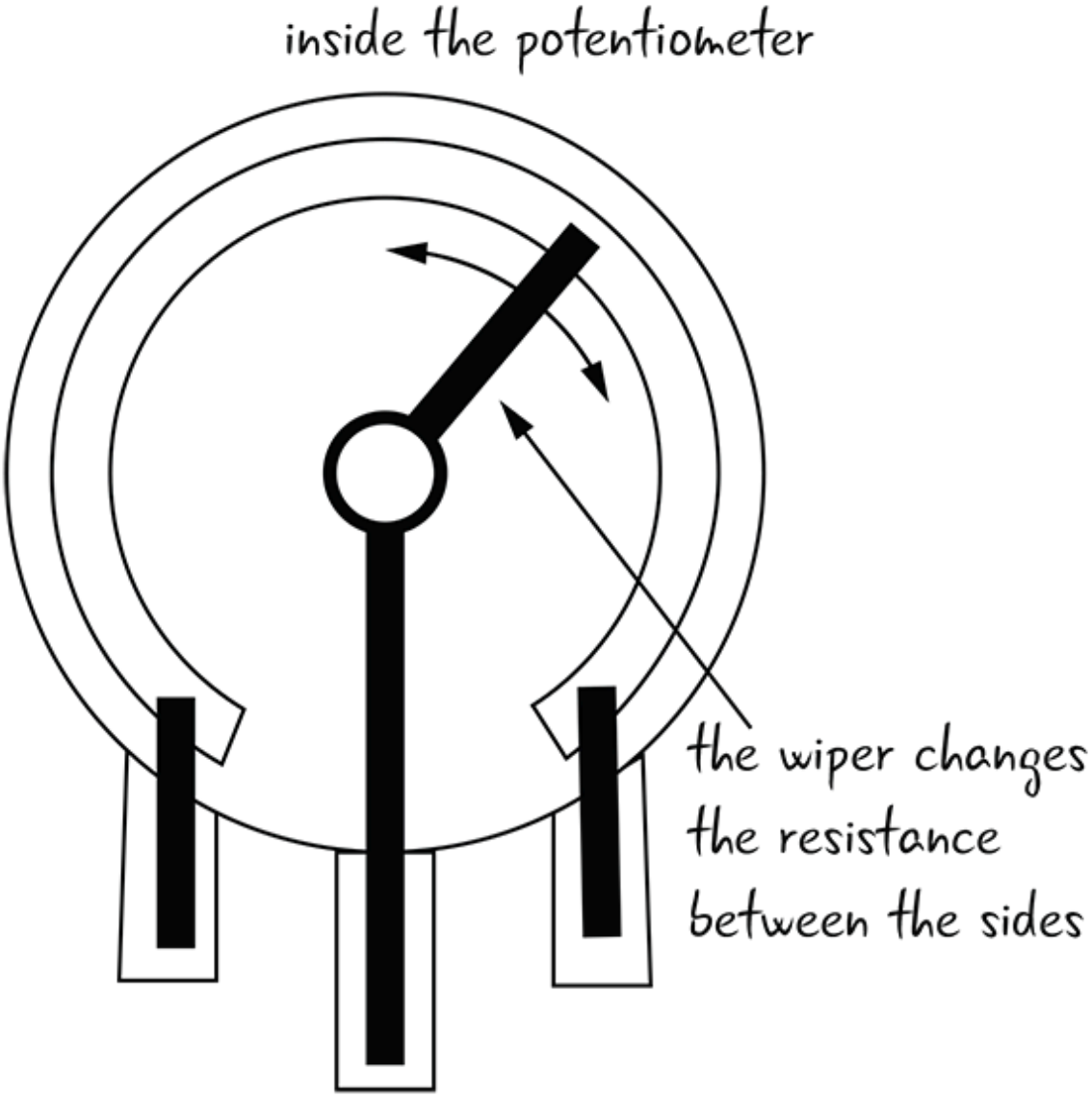


FIGURE 7-33: X-ray of potentiometer

PLAYING THE THEREMIN

You can play your light theremin by moving your hand closer to and farther away from the photoresistor. Changing the amount of light that falls on the photoresistor changes the resistance. Move your hand up and down to hear the eerie tones changing pitch (Figure 7-34).

You might try shining a flashlight on the photoresistor (Figure 7-35); the pitch should jump up as you reach the highest light level.

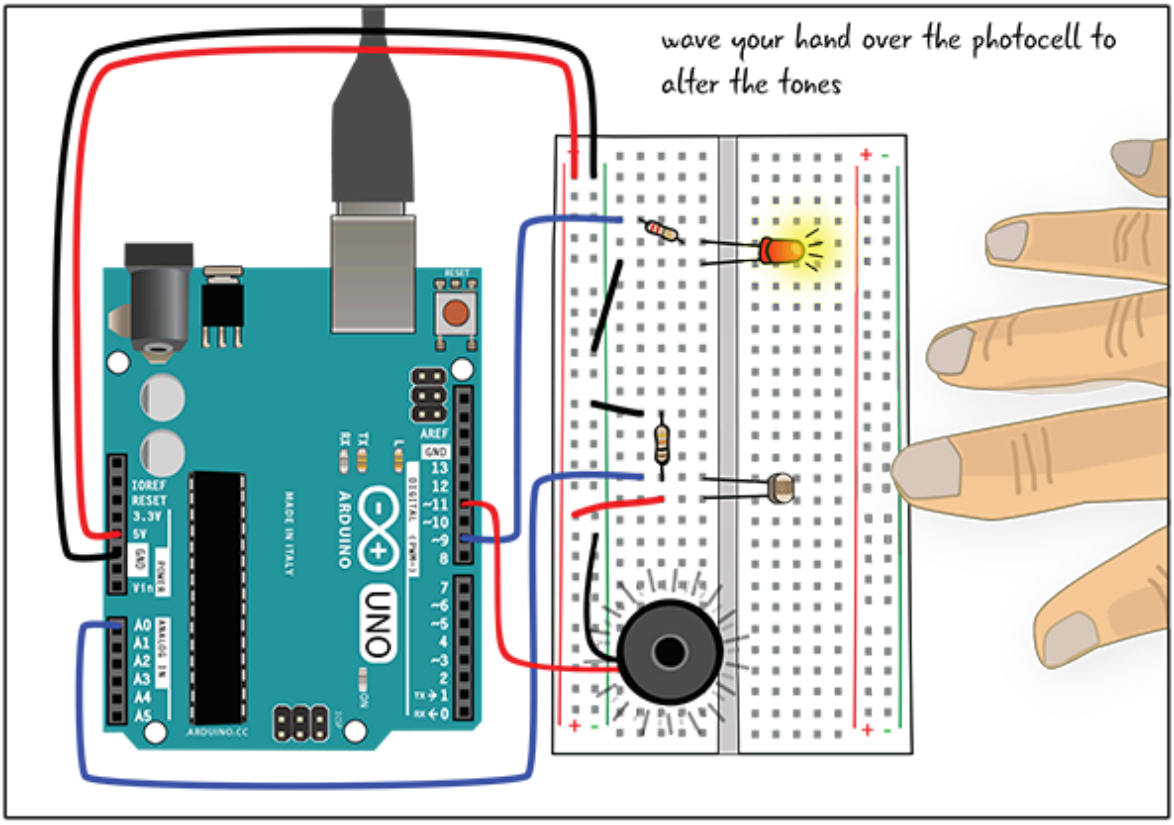


FIGURE 7-34: The pitch changes as the photoresistor is exposed to different amounts of light.

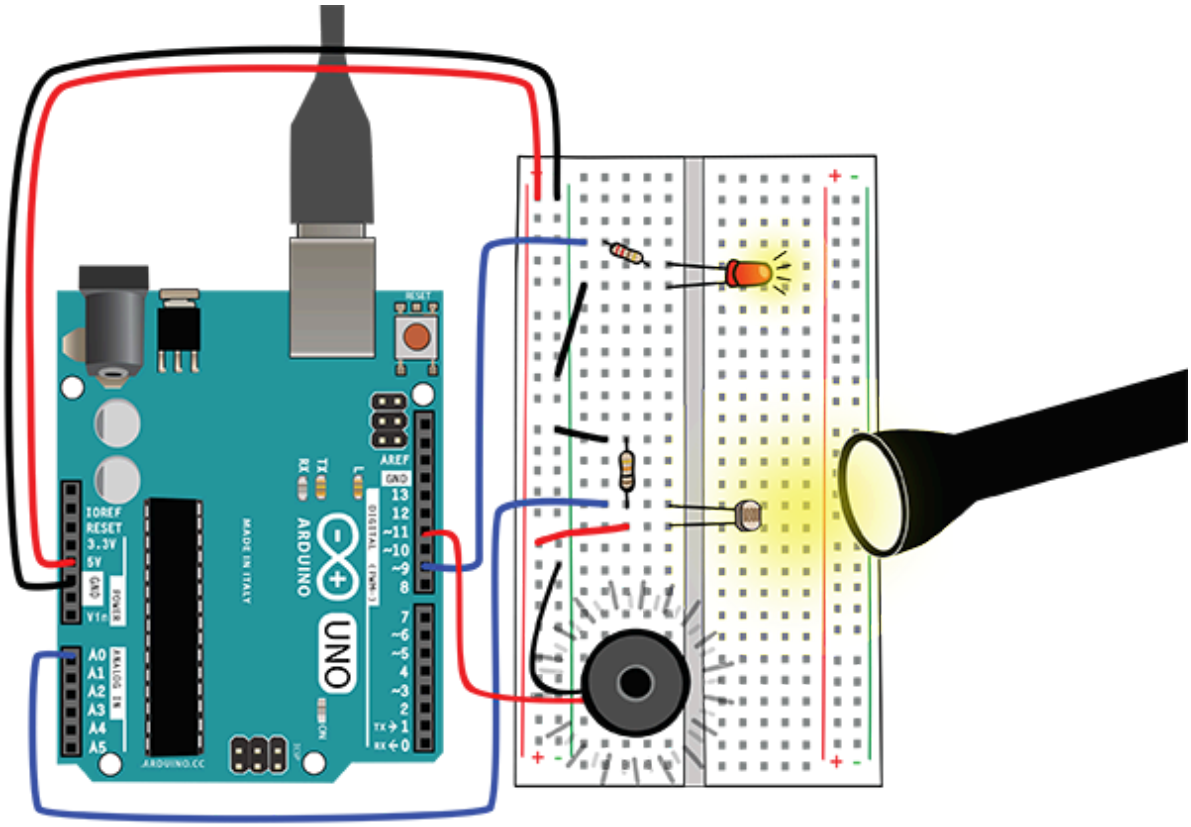


FIGURE 7-35: Shining a flashlight on the photoresistor

WHY DIDN'T THE CODE CHANGE?

You didn't need to change the code when you replaced the potentiometer with the photocell. How can this be? As described earlier, the photoresistor works on the same basic principle as the potentiometer. Both types of variable resistors change the values of the resistance in the circuit, which, as you know from Ohm's law, alters the value of the voltage (and the current as well) on the Arduino. The code you've written for your light theremin will work with a potentiometer, a photoresistor, or any other variable resistors you want to use.

READING THE SERIAL OUTPUT

The serial window displays the values sensed by your photoresistor, but what do these numbers mean? More light shining on the

photoresistor creates a lower resistance and consequently a higher sensor value (Figure 7-36).

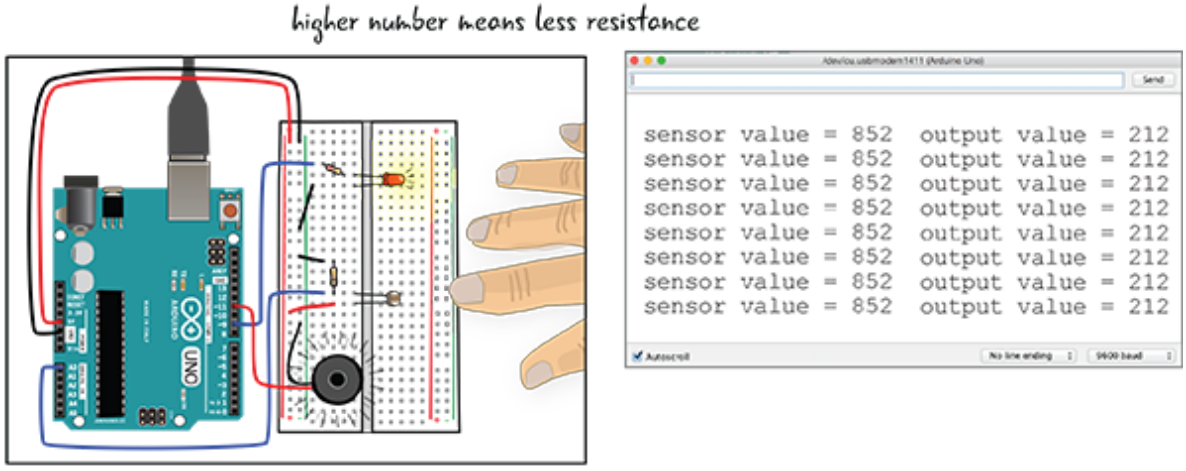


FIGURE 7-36: More light means a lower resistance value.

If the photoresistor detects less light, the resistance value of the sensor is higher, and the number in the serial monitor will be lower (Figure 7-37).

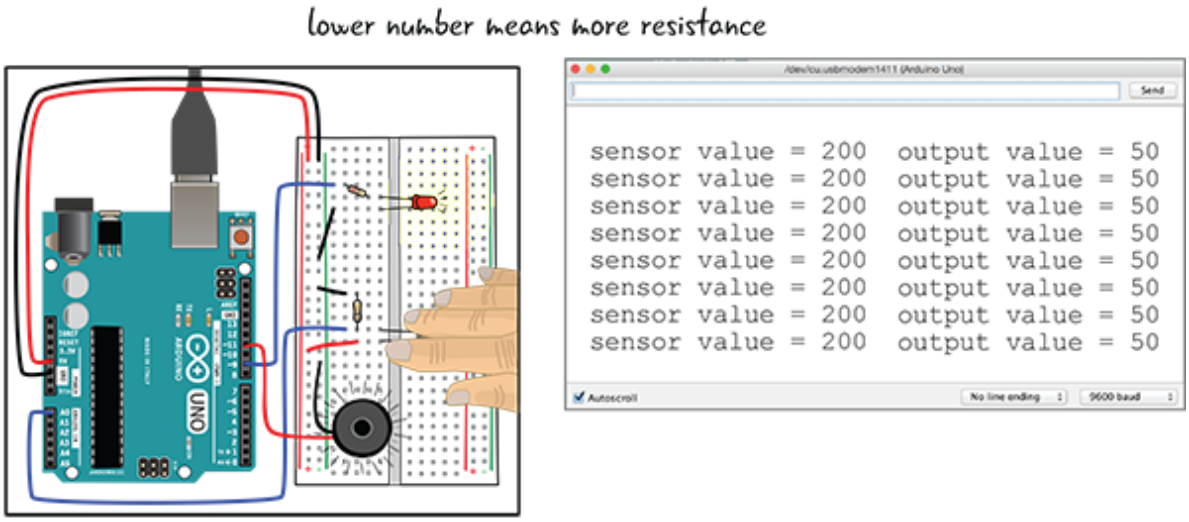


FIGURE 7-37: Less light means a higher resistance value.

It's good practice to get used to reviewing the information displayed in the serial monitor. You may need it to troubleshoot problems.

SUMMARY

In this chapter, you learned how to attach a potentiometer and a photoresistor to the analog input pins in the Arduino to get a range of values to use in your sketches. You learned what PWM means and how the Arduino uses the PWM pins with `analogWrite()` to simulate an analog output. You now know how to map values from the range you receive from your inputs to a range that is appropriate for the output you are using. And you learned to use the serial monitor in the Arduino IDE to read values from inputs. In the next chapter, you'll build on this knowledge by creating a circuit that turns motors. Download the code for LEA7_VariableResistorTone here: github.com/arduinotogo/LEA/blob/master/LEA7_VariableResistorTone.ino.

8

SERVO MOTORS

In this chapter, you'll add motion to your Arduino projects. You will be using servo motors, as shown in Figure 8-1.

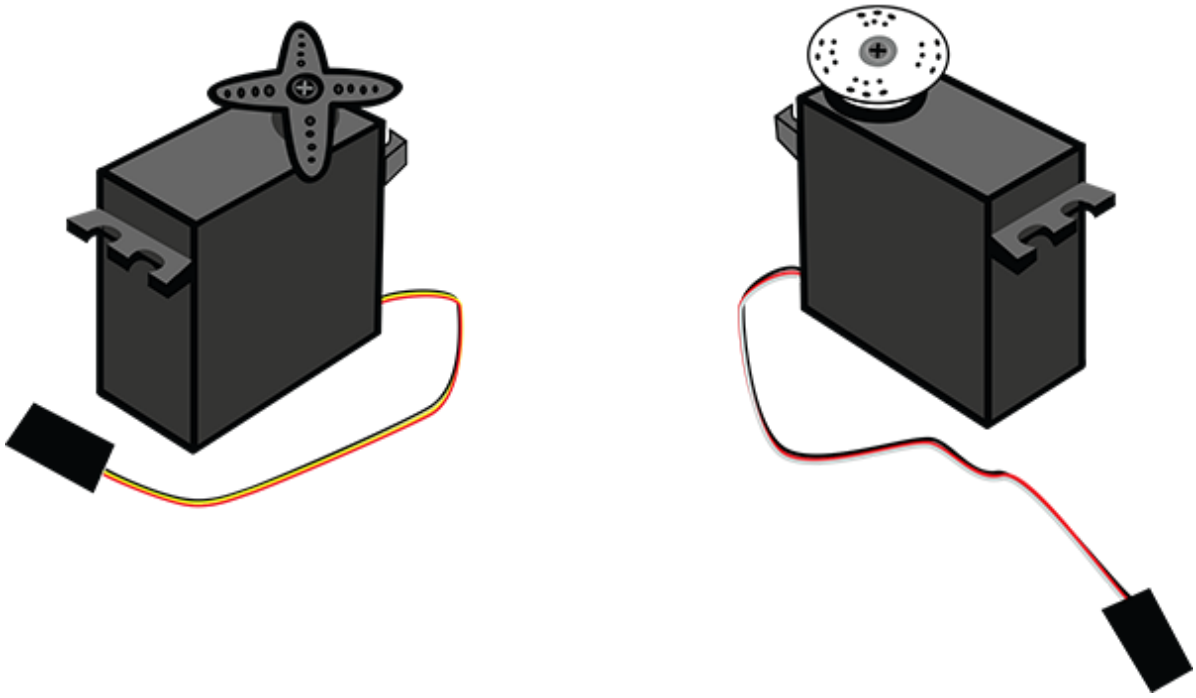


FIGURE 8-1: Hobby servo motors

Servo motors are a type of motor that can be easily programmed to rotate to a precise position. A servo motor contains a set of gears

and a control mechanism that rotates a shaft a specified number of degrees. Because servos are relatively easy to control, they are a good introduction to using motors in your projects. Although there are many types of servos, the ones that we recommend you use can rotate between 0 and 180 degrees.

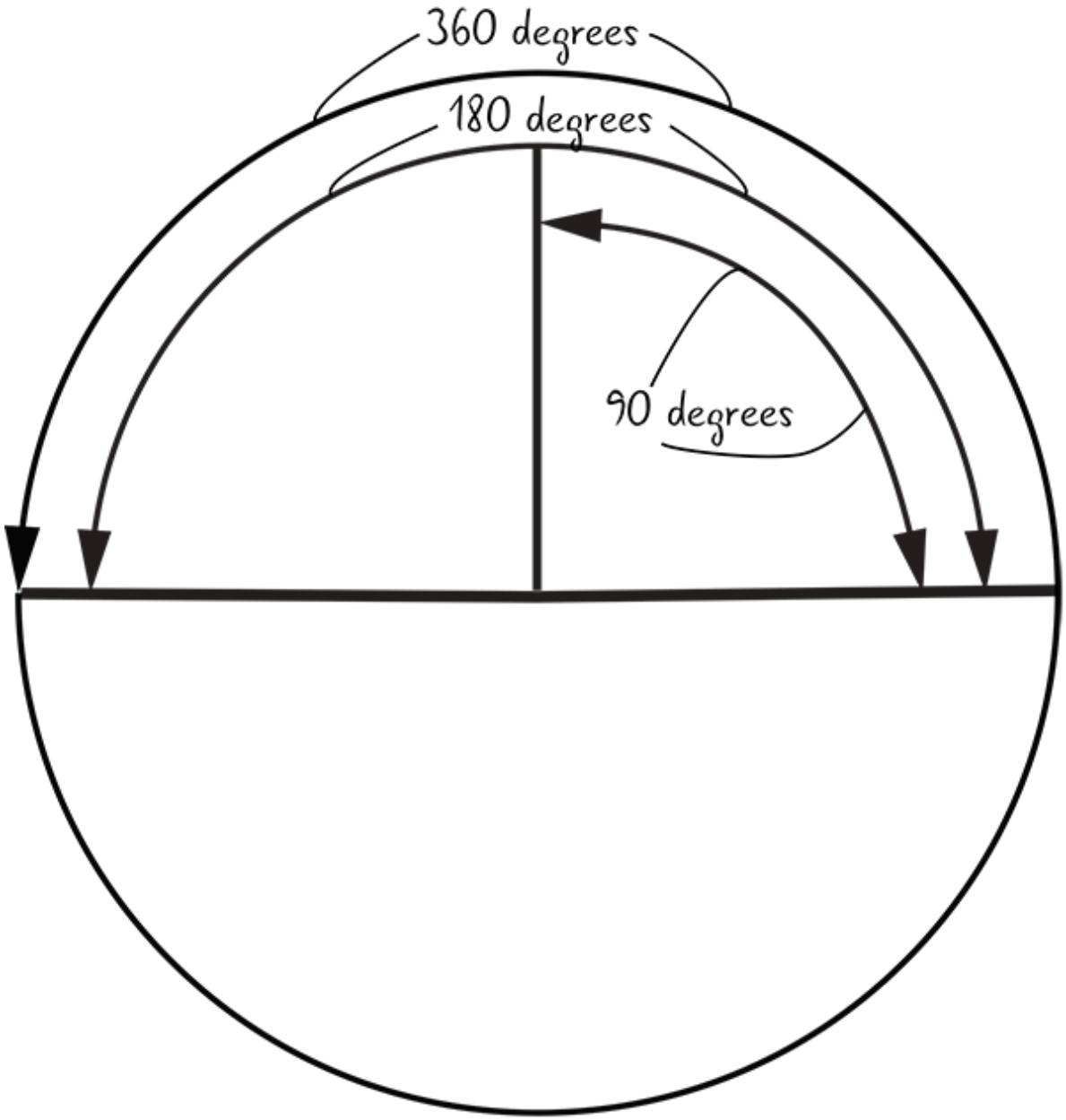


FIGURE 8-2: Diagram of degrees of rotation

First, you'll be turning your servo continuously with an example sketch from the Arduino IDE. Then, you'll control a servo with a potentiometer. Finally, you'll add a second servo to the circuit and adapt a sketch so that the movement of both servos is controlled by turning the potentiometer.

We'll also cover some programming concepts that you haven't encountered before, including `for` loops and custom functions.

The type of servo motors you'll be working with are called *positional rotation servos*. They are limited to 180 degrees, or one half of a full rotation of movement; degrees of rotation are shown in Figure 8-2 (previous page). They are accurate to any degree within that range, meaning if you need the shaft of your motor to point to an exact spot, they are a great fit.

Servo motors are used in a wide variety of applications, including hobby model airplanes, robotics, and art projects of all shapes and sizes.

WAVING THE FLAGS

Figure 8-3 shows a drawing of the first project you're going to build. We'll review a bit about analog data, look more closely at servos, and then get started building.

ANALOG DATA REVIEW

You learned in the previous chapter that analog data can refer to any information that has more than the two possible values that digital information can hold (described alternately as 1 or 0, true or false, `HIGH` or `LOW`). In your Arduino sketches, you saw that the number of possible values was often mapped to a particular range—for inputs, a value between 0 and 1023, and for outputs, a value between 0 and 255. Having a wider range of values allows you to do more than just turn your components on or off.

Servo motors use precise positioning. You'll use analog data in order to set the direction the shaft of your motor is facing in the projects you build in this chapter.

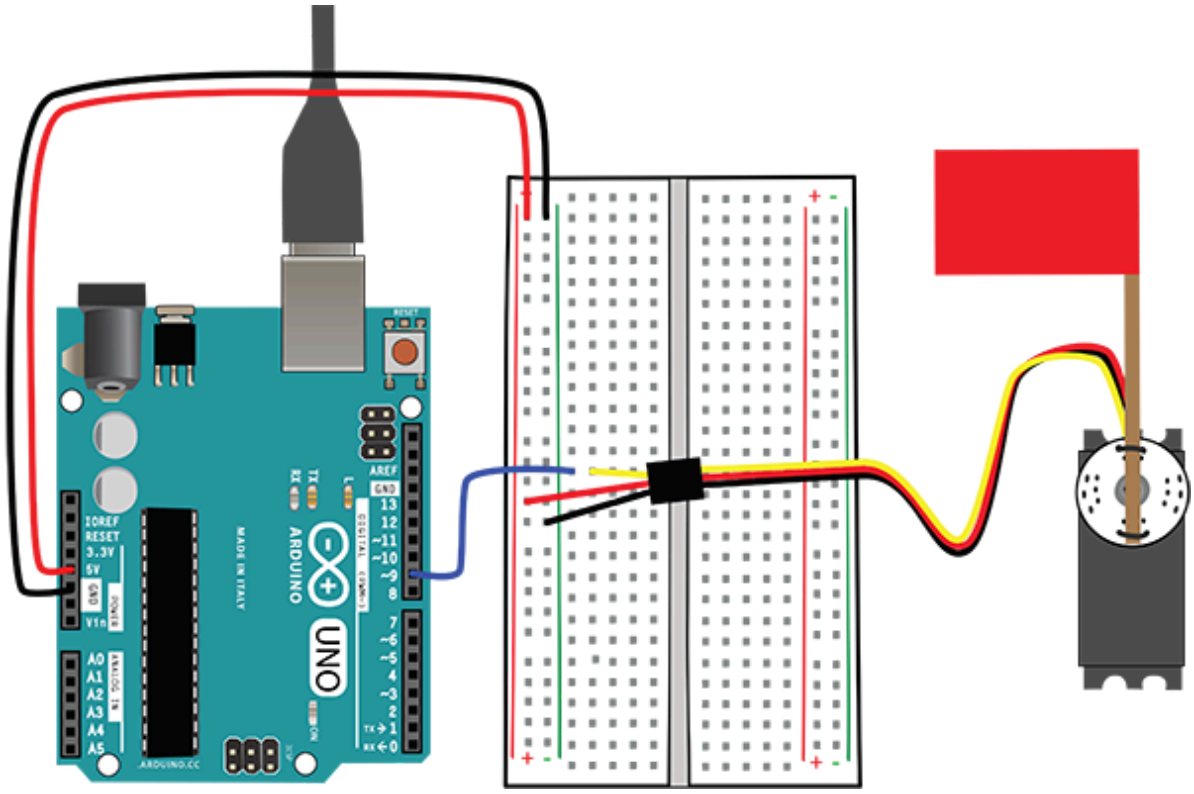


FIGURE 8-3: The servo will turn and wave the flag.

SERVOS UP CLOSE

As we have said, many types of servo motors are available. We recommend that you use a motor with a range of 180 degrees that runs on 4.8V to 6V. This type of standard servo is commonly available from many online vendors or from hobby shops or stores that sell electronic components.

PARTS OF A SERVO

The mechanisms that turn the servo (motor, gears, and circuit) are enclosed in a case. The spline is the part of the movable shaft that extends through the case. The horn, or arm, attaches to the spline. A

screw holds the horn in place on the spline. The packet of mounting materials that comes with your servo will generally have a variety of horns that you can attach so that you can switch them depending on the nature of your project, as well as some screws and other fasteners. The servos are designed to make it easy to unscrew a horn and replace it with another one. Servos also generally have mounting flanges on the front and back, making the servo easy to attach to your projects.

ONLINE VENDORS

Online vendors include the following:

adafruit.com

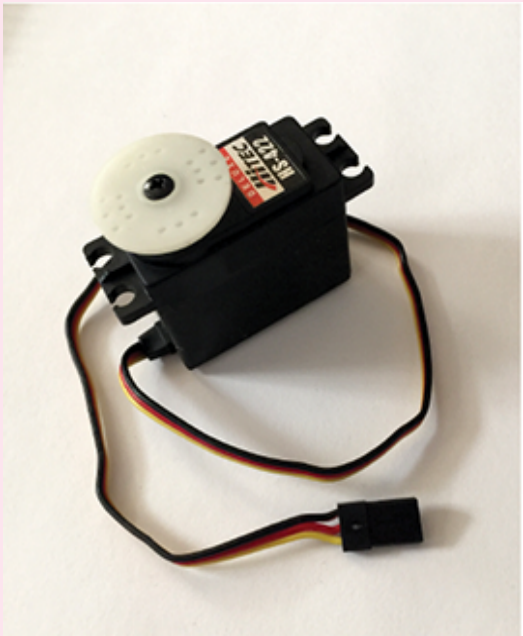
sparkfun.com

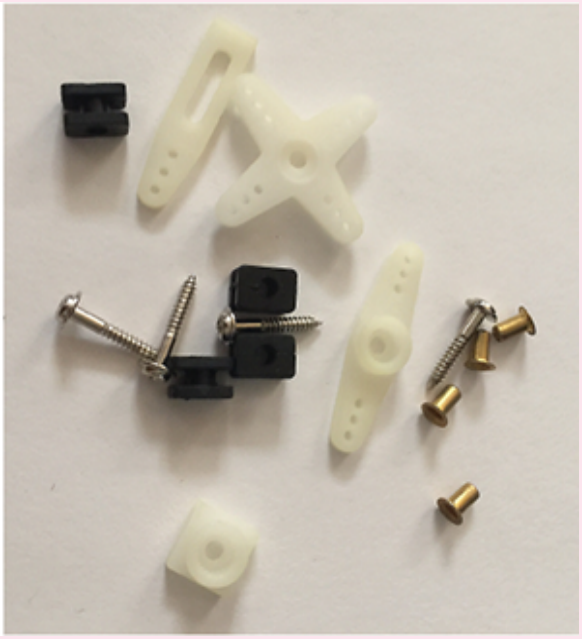
makershed.com

microcenter.com

servocity.com

When you purchase a standard servo motor, you will receive the motor and a package that contains mounting hardware, as shown here:





A cable is connected to the front of the case near the bottom. This has three color-coded wires; the black wire will be attached to ground, the red wire will be attached to power, and the third wire, sometimes yellow, sometimes blue, sometimes white, is the control

wire. You will be connecting the control wire to a pin on the Arduino. The servo has a plug, or connector, at the end of the cable to attach it to a circuit. Figure 8-4 shows a servo with and without a horn attached.

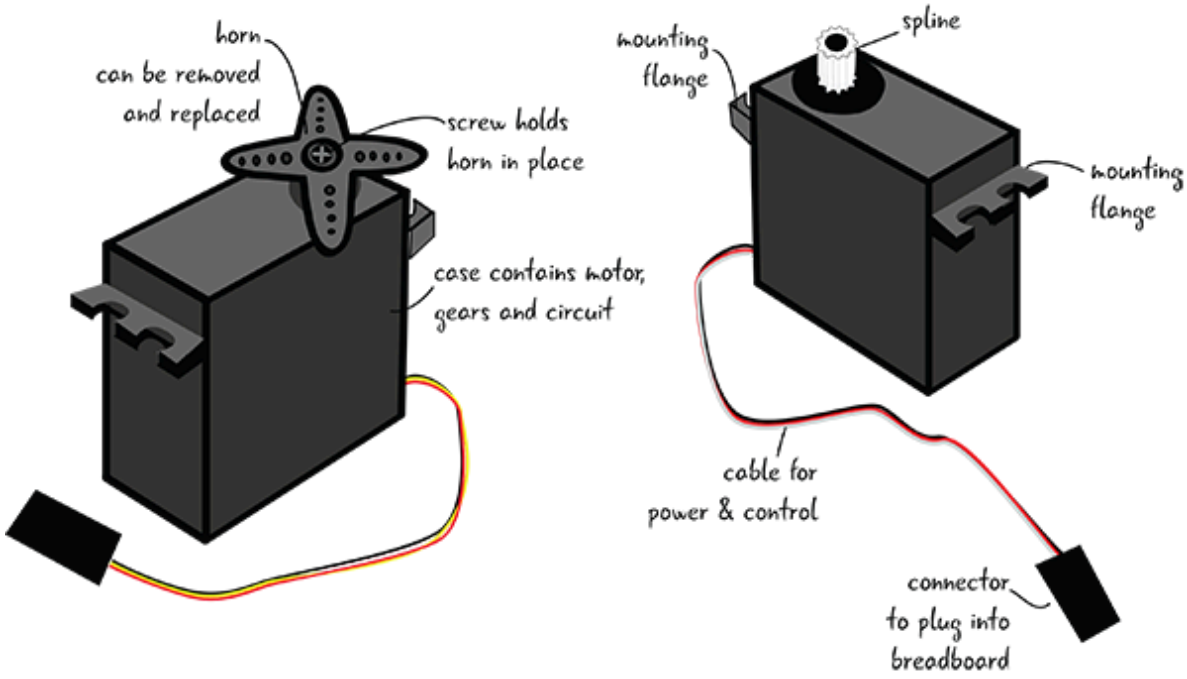


FIGURE 8-4: Servos annotated, one with the horn and one with the horn detached

The different styles of horns that come with your servo allow you to attach the correct horn for the project you are building (Figure 8-5).

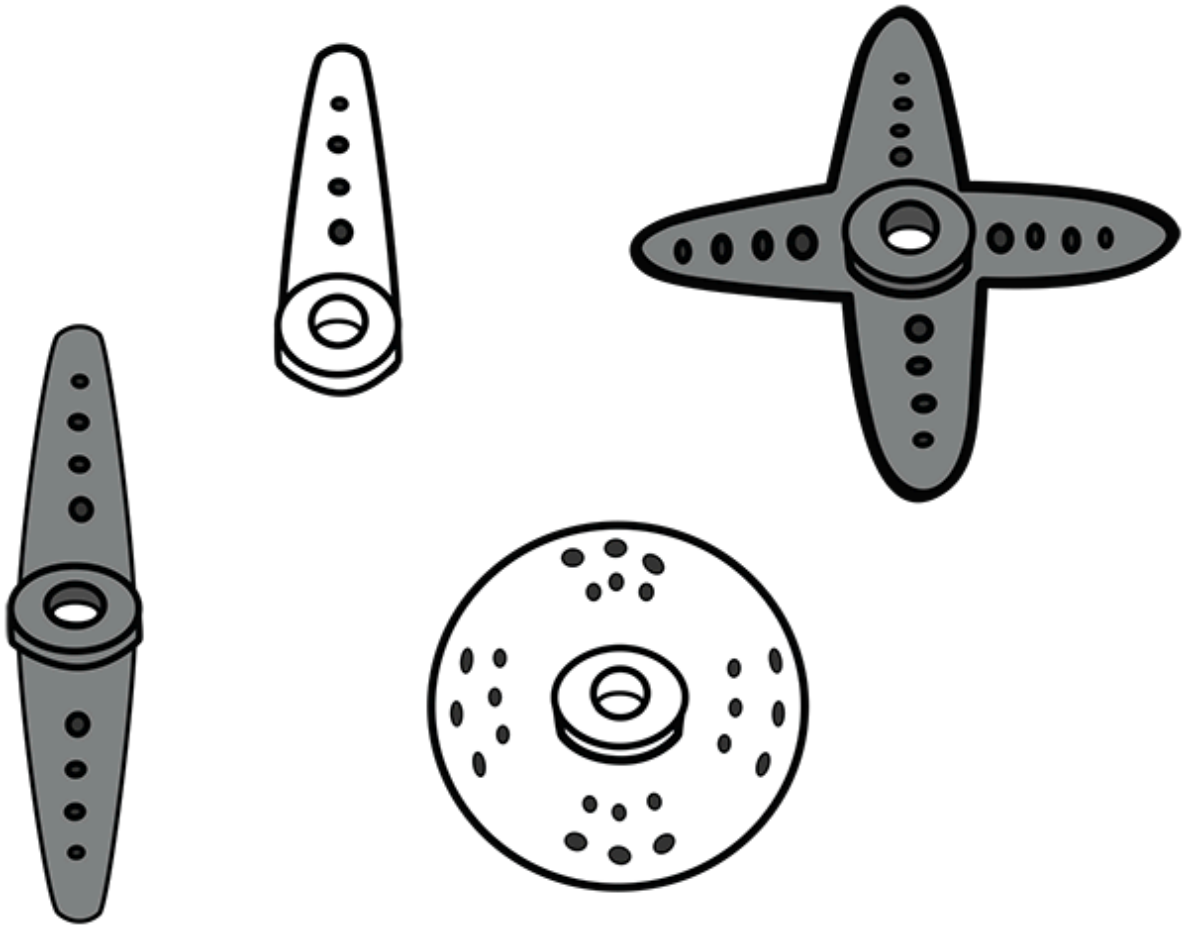


FIGURE 8-5: Some servo horns

QUESTIONS?

Q: Why are we starting with servo motors rather than another type of motor?

A: We are starting with servo motors because they are easy to control and wire.

Q: Will I need any other type of motors for my projects?

A: Yes, though servos are useful, they will not work for every project. Sometimes it is appropriate to use a DC or a stepper motor because of power requirements, or because of the particular task that they are to perform. They are attached to the circuit and programmed differently, which we will not be covering in this book.

Q: My servo wires don't match the colors you mentioned. Which wires go to power and ground?

A: On some servos, the ground wire is brown; generally power is red on most hobby servo motors. Look at the front of the servo to see how the wires are coming out on the cable. Generally the ground wire is on the right, the power wire in the middle, and the control wire on the left.

BUILDING THE SERVO CIRCUIT STEP BY STEP

You'll need these parts:

Standard servo motor

Breadboard

Jumper wires

Wooden coffee stirrer or strip of cardboard

Tape

Colored paper

Arduino Uno

USB A-B cable

Computer with the Arduino IDE installed

Figure 8-6 shows the schematic and the drawing of the first circuit you are going to build. As usual, the power and ground buses on the breadboard are attached to 5V and GND on the Arduino. You can see that the servo has three wires: one attached to power, one to ground, and one to a pin on the Arduino.

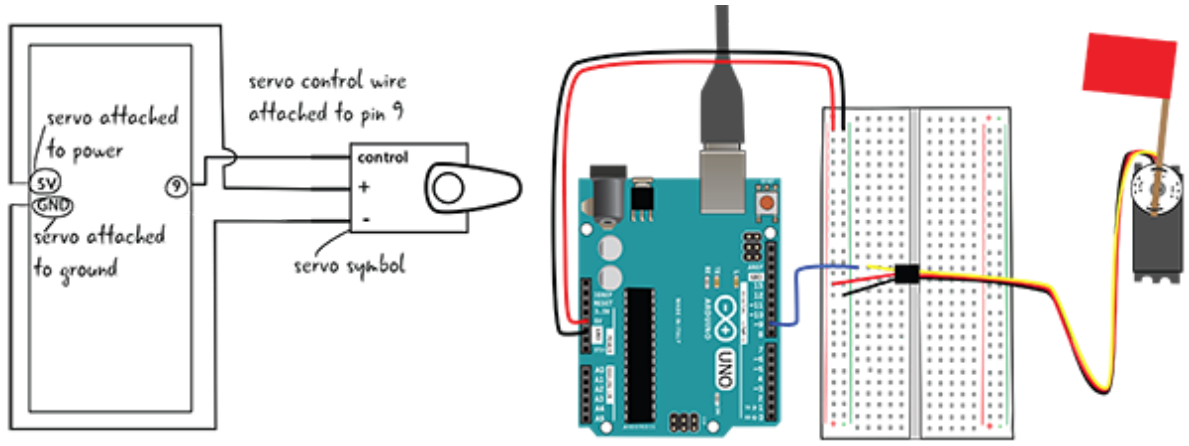


FIGURE 8-6: Schematic and drawing of our first servo circuit

There are a few things you should know about the servo that will make it easy to set up.

PREPARING THE SERVO

You've seen that the servo comes with a packet of different horns. You might want to swap out the horn that is attached to your servo

when you purchase it. Use a small screwdriver to remove the screw attaching the horn and replace it with another, as shown in Figure 8-7. We are using the circular horn in our examples.

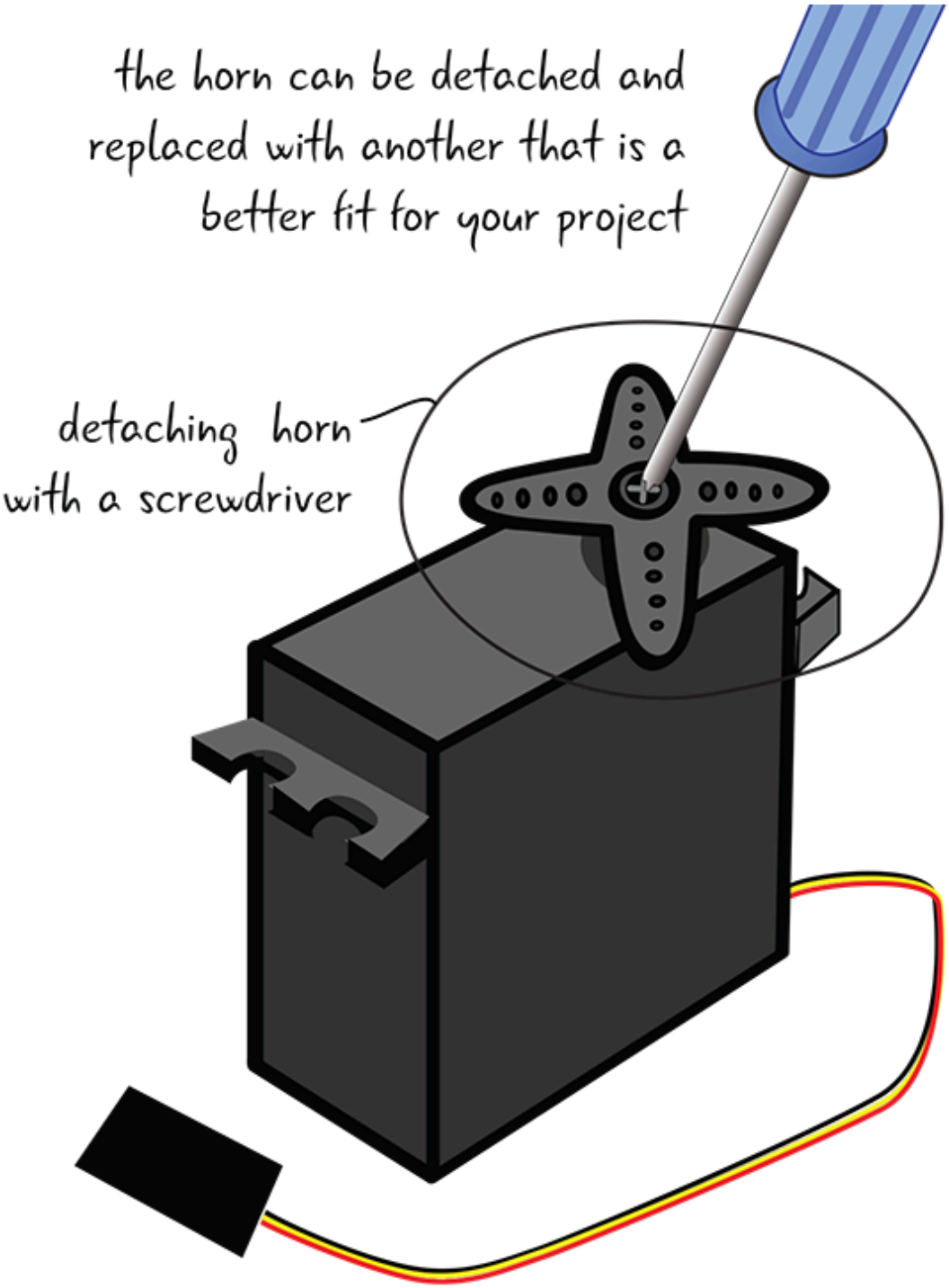


FIGURE 8-7: Removing the horn

We have made a flag with a coffee stirrer and a piece of colored paper. A strip of cardboard with a piece of colored foam would also

work. Make a flag with whatever materials you have lying around and attach it to your horn with wire, as shown in Figure 8-8.

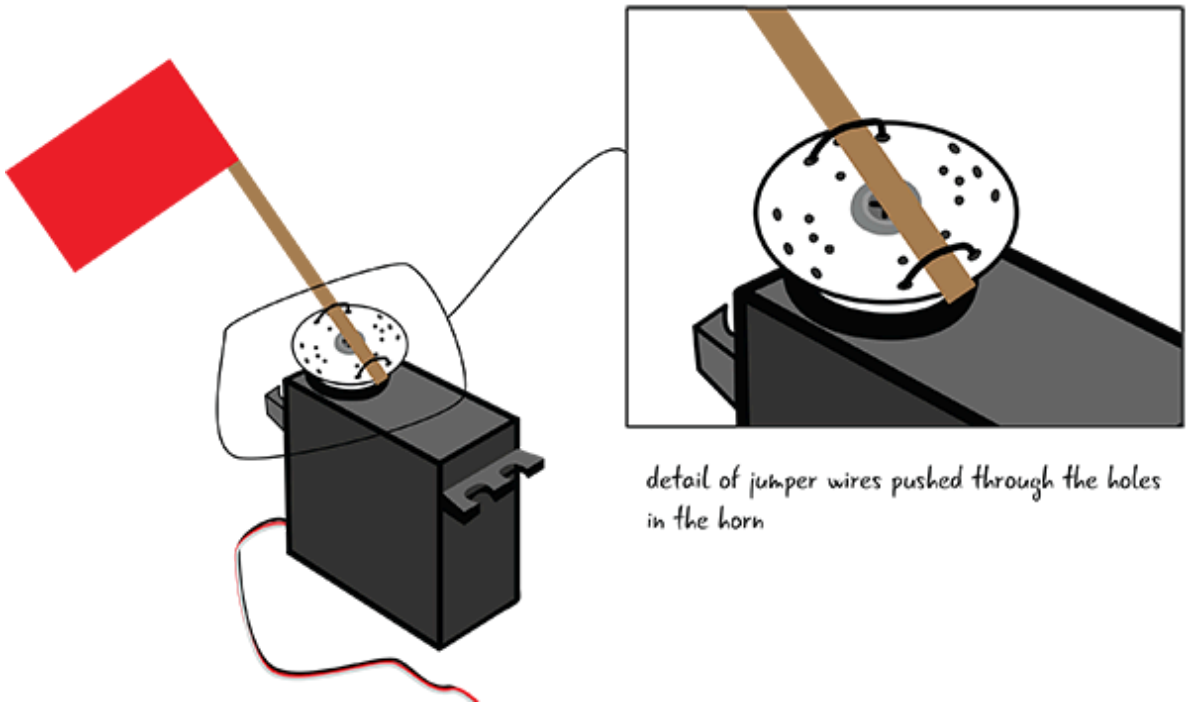


FIGURE 8-8: Attaching the flag to the servo horn

Before you attach your servo to the breadboard, you will have to add jumper wires to the plug/connector on the servo. As you know, there is a control wire, a wire that goes to power, and one that goes to ground. Follow the color conventions (red wire connected to power, black to ground) as usual. If you have a jumper that's the same color as your control wire, use that, or use a color that is distinctive from the red and black wires. In our example (Figure 8-9), the control wire is yellow, but sometimes that wire will be another color, such as white.

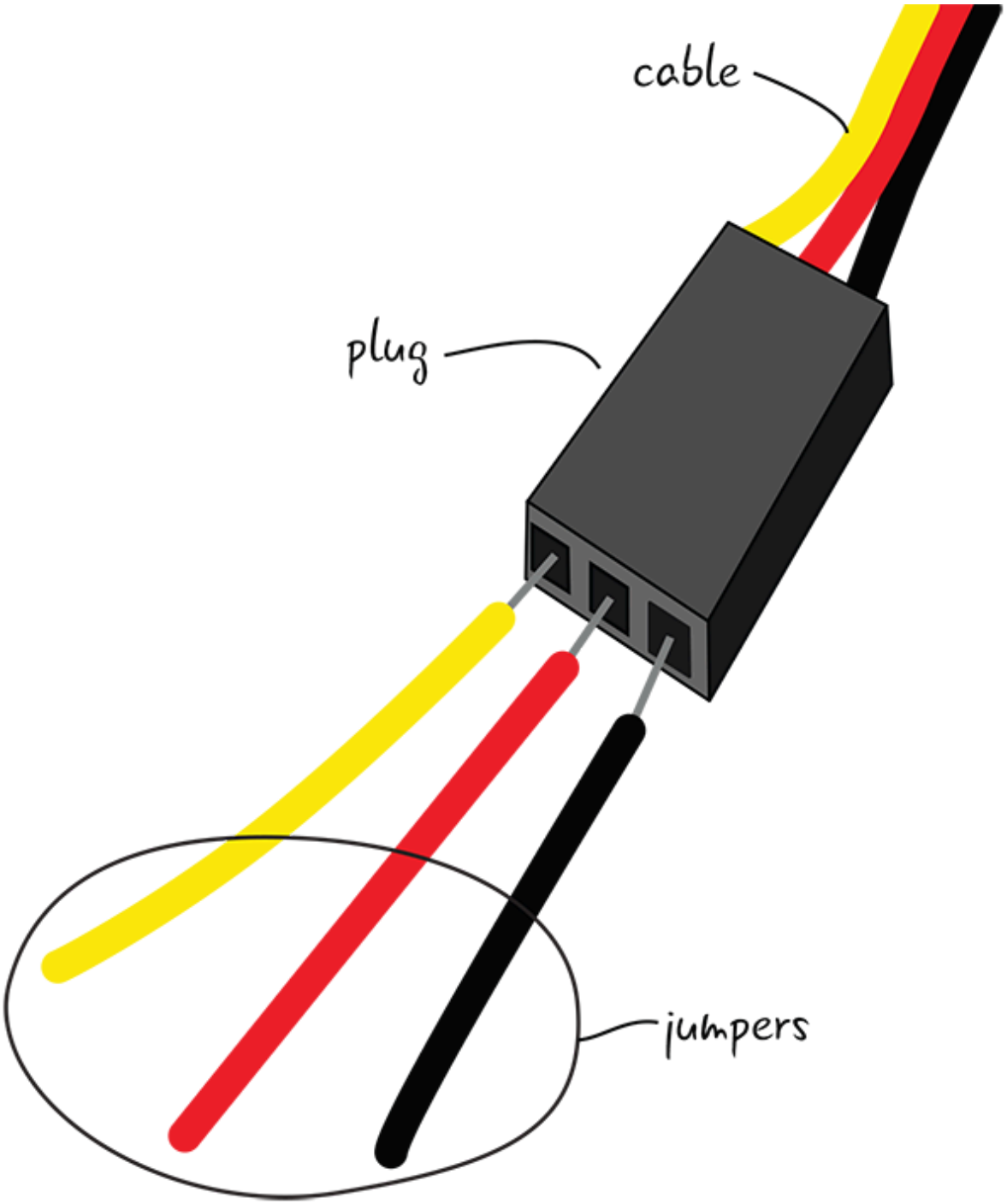


FIGURE 8-9: Servo connector up close

ATTACHING THE SERVO

Attach jumpers from GND on the Arduino to the ground bus on the breadboard and from 5V on the Arduino to the power bus.

Now attach your servo to the breadboard. Attach the red power jumper to the power bus, and the black ground jumper to the ground bus. Then place the control jumper wire in its own row of tie points (Figure 8-10).

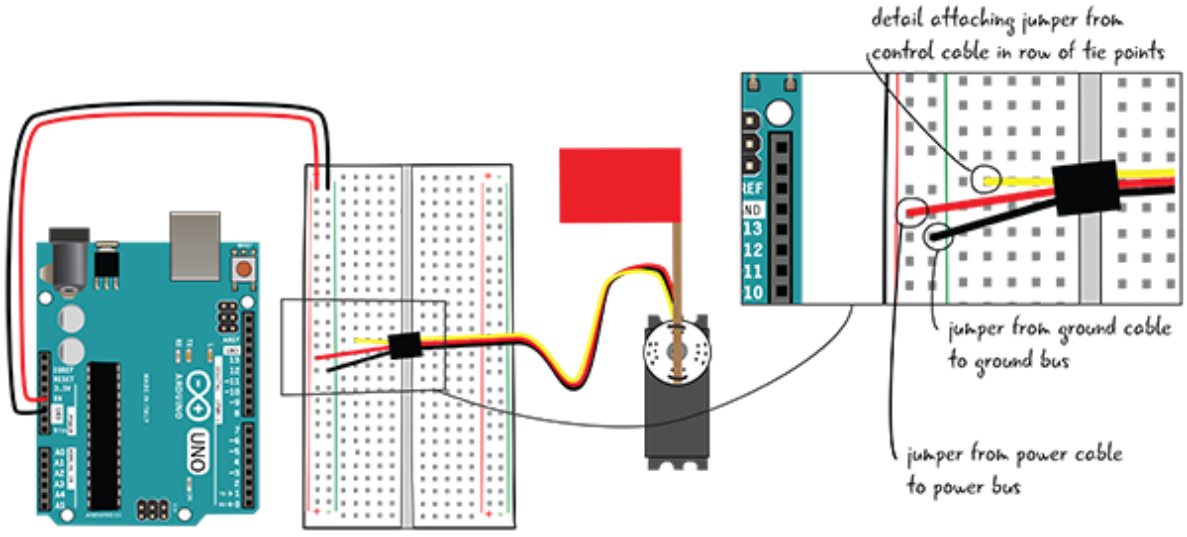


FIGURE 8-10: Attaching the servo to the breadboard

Next, attach a jumper from Pin 9 to the same row of tie points as the jumper from the control cable, as seen in Figure 8-11. You are attaching the servo to Pin 9 because that is the pin it's attached to in the sketch you will be downloading. It could be attached to any of the digital pins, 2 through 13.

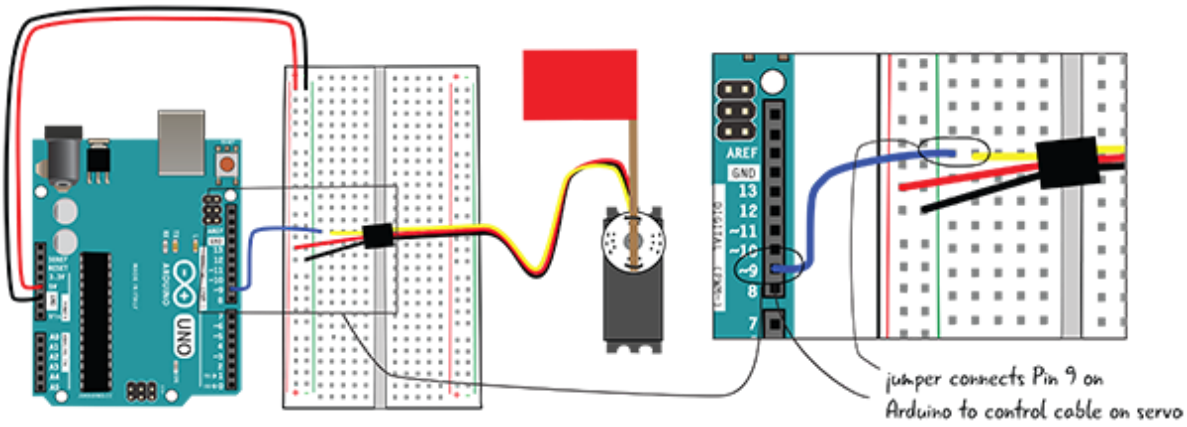


FIGURE 8-11: Attach the control wire of the servo through a jumper to Pin 9 on the Arduino.

You're now ready to download the sketch from the Arduino IDE.

ATTACH YOUR COMPUTER AND DOWNLOAD THE SWEEP SKETCH

Now that you have completed wiring your circuit, you need to download a sketch to your Arduino in order to run your servo. The Arduino includes a few sketches about using the servo motor, and for this first example, you'll use the Sweep sketch included in the Servo folder of example sketches (File > Examples > Servo > Sweep).

When you have opened the sketch, save it as LEA8_Sweep. If you haven't already done it, attach your computer to your Arduino and upload the sketch.

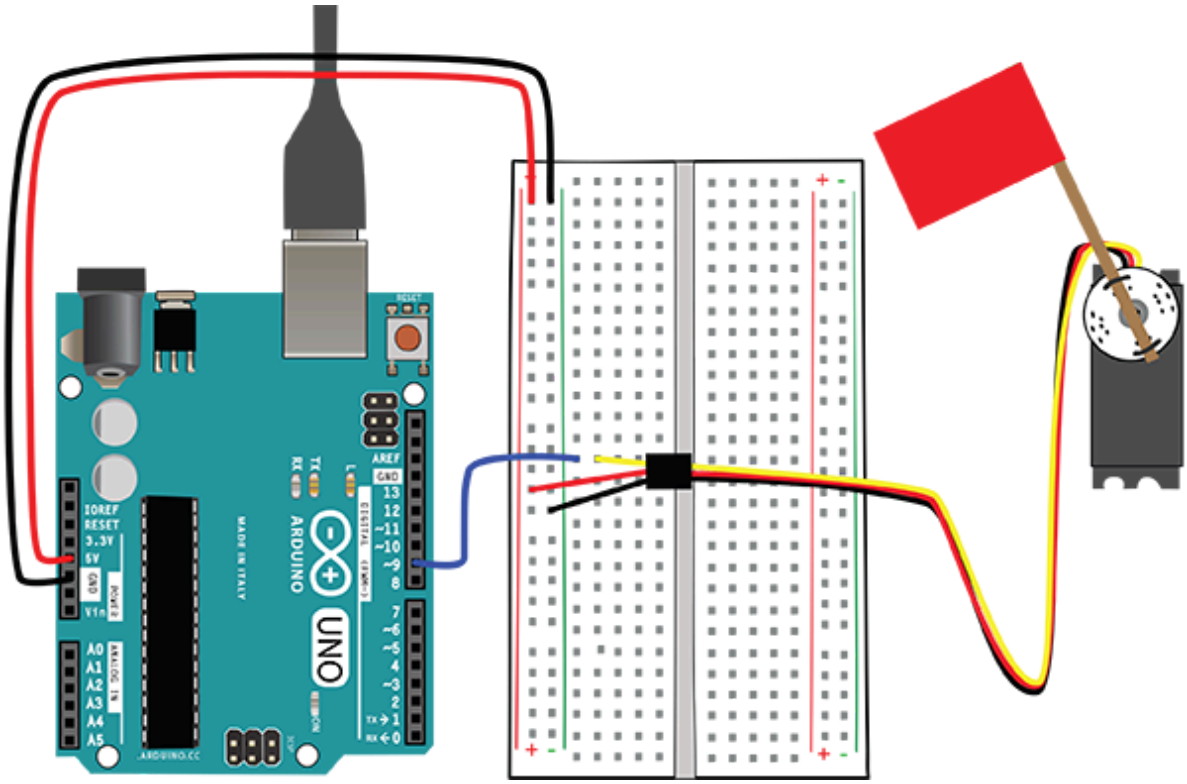


FIGURE 8-12: The flag waves.

Wave the Flag!

You should begin to see the servo motor swing the flag attached to the horn 180 degrees in one direction, and then reverse directions and swing back to its starting position (Figure 8-12). It will continue to loop these movements one after another for as long as the Arduino has power. Let's take a closer look at the code and explain what each line is doing.

LEA8_SWEEP OVERVIEW

In our breakdown of some of the sketches in this chapter, we have removed the comments section for readability. Figure 8-13 shows a quick look at the sketch.

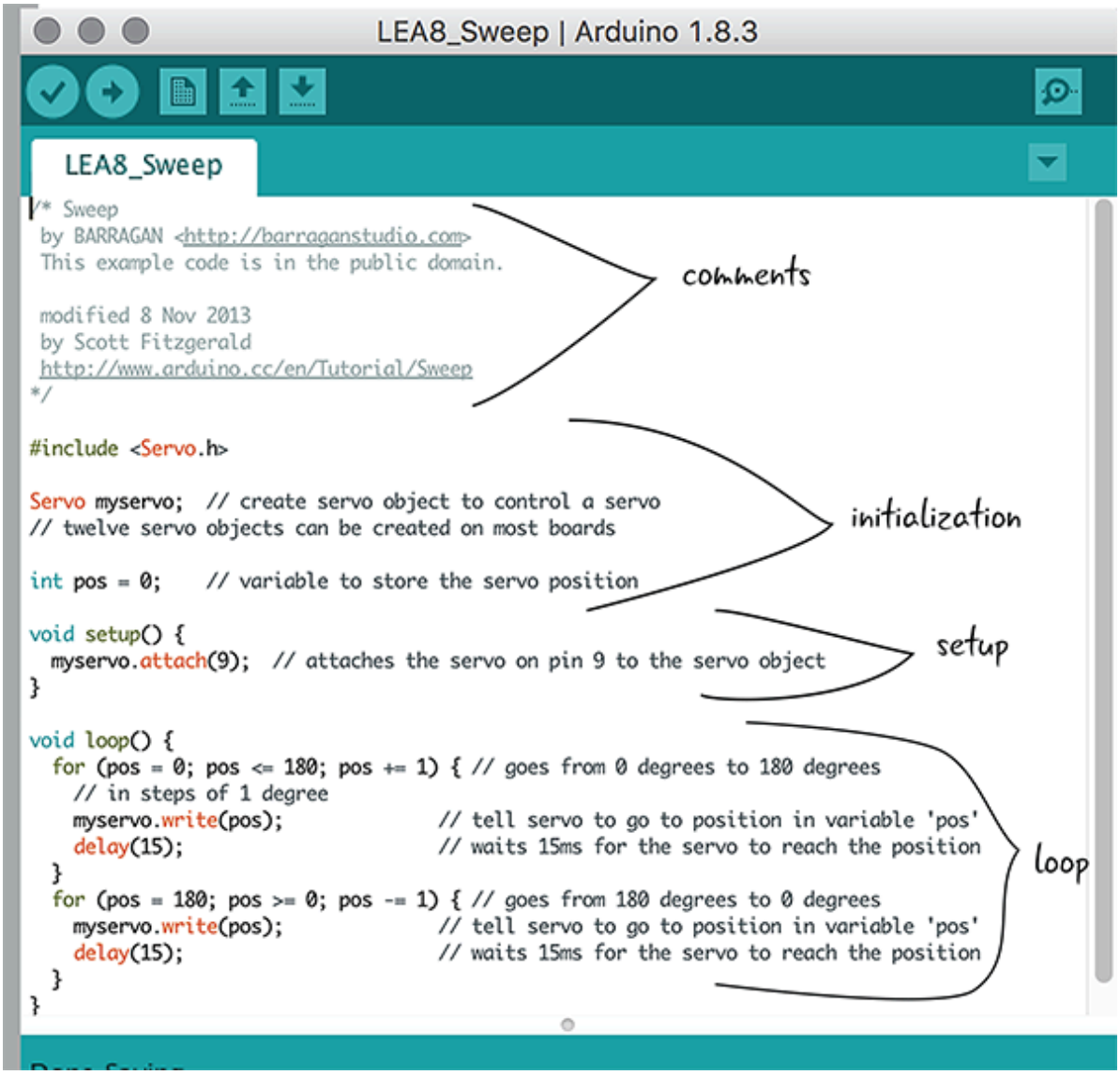
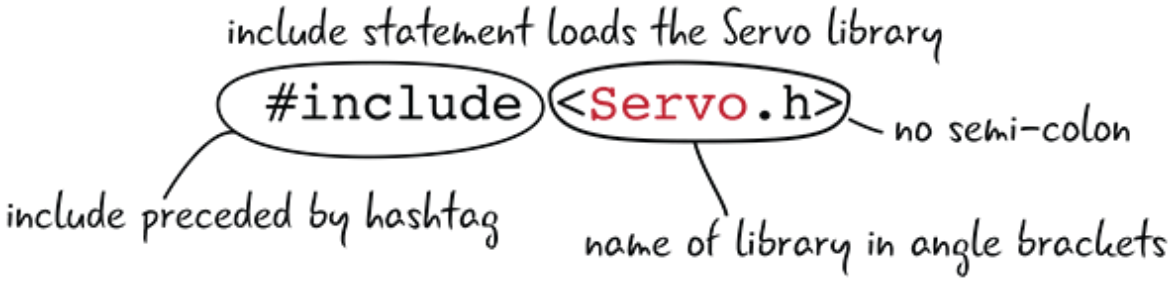


FIGURE 8-13: LEA8_Sweep overview

INITIALIZATION

The first thing you see in the initialization section is a line of code that is going to add functionality to your Arduino. The include statement tells your Arduino to load a *library,* which will extend the capabilities of your Arduino. Rather than having to write all the code yourself, including libraries gives you access to extra functions that other people have written that expand the possibilities of the Arduino.

How do you add a library? We use an include statement, which starts with a `#` followed by the word `include`. An open angle bracket follows, with the name of the library, in this case `Servo`, and the extension `.h` next. A closing angle bracket completes the statement, and there is no semicolon in this instance.

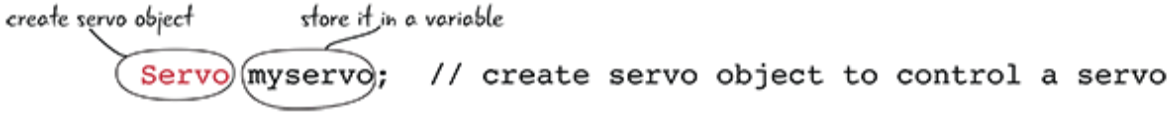


Note

A *library* is a set of code that extends the functionality of an Arduino. The library must also be specifically included in your sketch in order to use it.

The Arduino IDE has many libraries already loaded, and it also allows you to load new libraries if you want to have access to them. For now, we are just concerned with the Servo library and what it does.

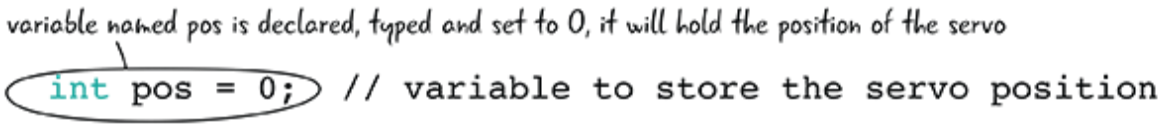
If you look at the next line of our initialization section, you see a new type named `Servo`. After loading the Servo library, you can create a servo object, which has functions that allow it to control servo motors. This line creates a servo object and stores it in a variable named `myservo`.



We have not discussed objects before, and an in-depth discussion of objects is beyond the scope of this book. Think of an object as a template with a set of attached functions and properties—that is, you can create several different servo objects in your sketch based on the template. Although each one follows the same basic structure, you can modify their properties, such as position.

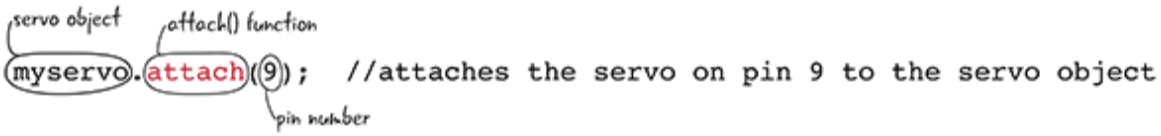
> **Note**
>
> An *object* is a template that includes properties and functions. Each instance of an object can have unique qualities; in this chapter, you will see more than one servo object.

The last line in our initialization sketch creates a variable named `pos`, which is set to 0. This variable will be used to set the position of the servo. If you change this value and again send it to the servo, the motor will update with its new position. You will see the code that changes these values within the `loop()` code.



INSIDE *SETUP()*

Our `setup()` section includes just one line in this sketch. `attach()` is a new function made available to your Arduino from the Servo library that allows you to connect the servo object that you named `myservo` to a pin on your Arduino. That way, whenever you refer to `myservo` you are referencing the pin to which you have attached `myservo`, and you will be able to control the servo that is attached to that pin. In this project, the servo motor is attached to Pin 9.



INSIDE *LOOP()*

This `loop()` code is a little different than what you have seen in the past, and will introduce you to our next programming concept: the *for loop.* Let's take a look at the code; then we will break it down.

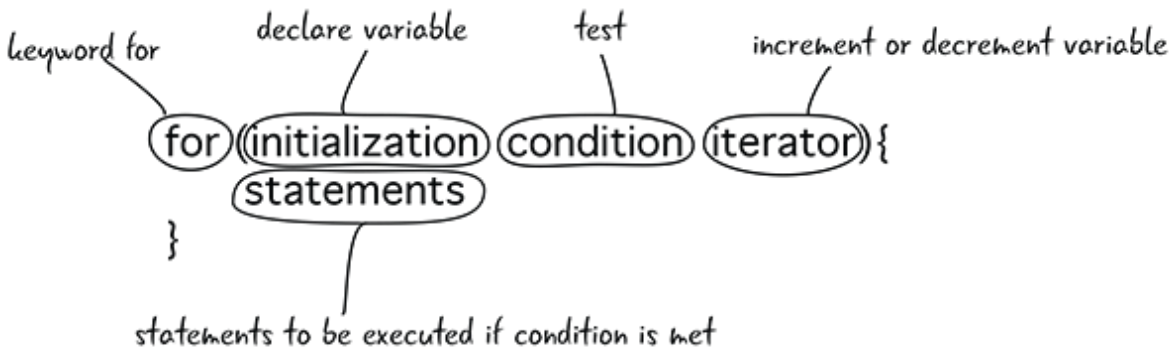
the for loop: code inside of loop()

```
for (pos = 0; pos <= 180; pos += 1) //goes from 0 degrees to 180 degrees
{                                   // in steps of 1 degree
    myservo.write(pos);     //tell servo to go to position in variable 'pos'
    delay(15);              //waits 15ms for the servo to reach the position
}
for (pos = 180; pos >= 0; pos -= 1) //goes from 180 degrees to 0 degrees
{
    myservo.write(pos);     //tell servo to go to position in variable 'pos'
    delay(15);              //waits 15ms for the servo to reach the position
}
```

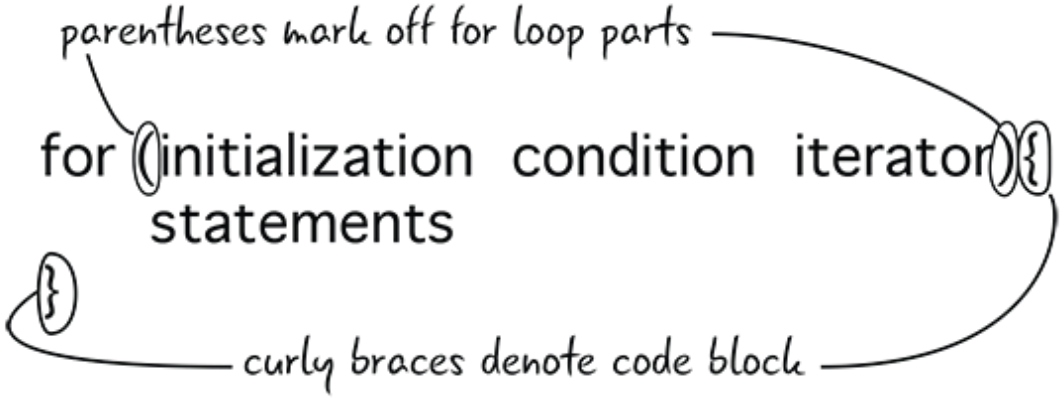
WHAT'S A *FOR* LOOP?

There are times when you might want to repeat something a certain number of times or until a particular condition is met. The `for` loop allows you to repeat something a number of times based on some conditions. In your sketch, you are setting the position of the shaft of the servo motor with a `for` loop.

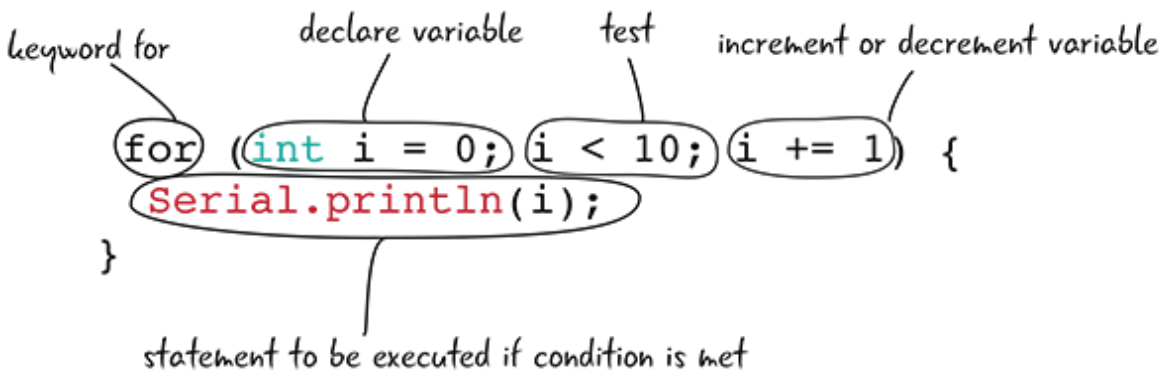
Let's first take a closer look at an example of a `for` loop before you see exactly what it does in your sketch. In the Arduino language, after the keyword `for`, the `for` loop has three parts: the *initialization,* the *condition* or test, and the *iterator*.



Here's how the parentheses and curly braces are used in a `for` loop. The parentheses mark off the initialization, condition, and iterator section. The curly braces mark off the block of code, or the statements to be executed if the condition is true.



Now let's look at an example of a `for` loop in the syntax of the Arduino language. This `for` loop prints integers from 0 to 9 in the serial monitor.



THINK ABOUT IT...

How would this work differently if you put the `for` loop in `setup()`? What about inside `loop()`?

HOW DOES A *FOR* LOOP WORK?

In what order do the parts of the `for` loop get executed? Let's take a look at Figure 8-14.

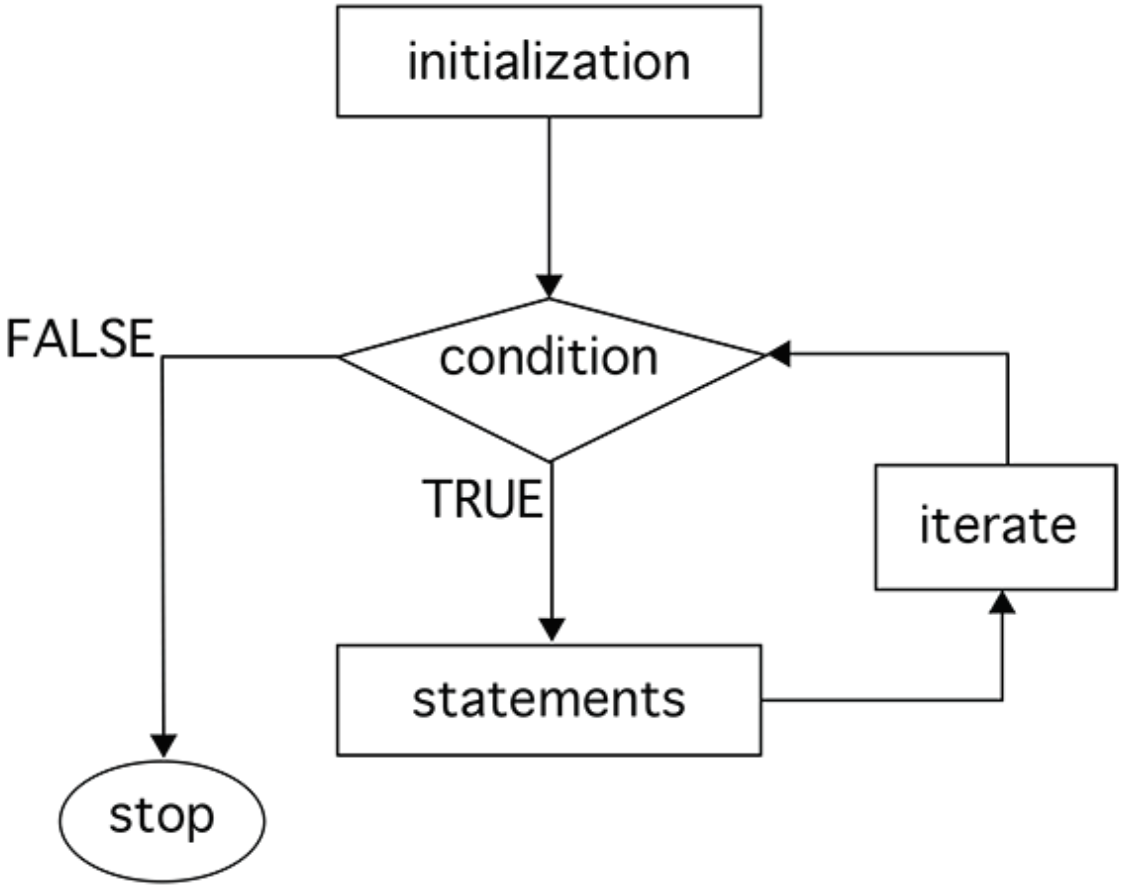


FIGURE 8-14: `for` loop flowchart

The first thing that happens in our `for` loop is the initialization (Figure 8-15). You create a temporary variable to count how many times you execute your `for` loop. The `for` loop will happen a certain number of times.

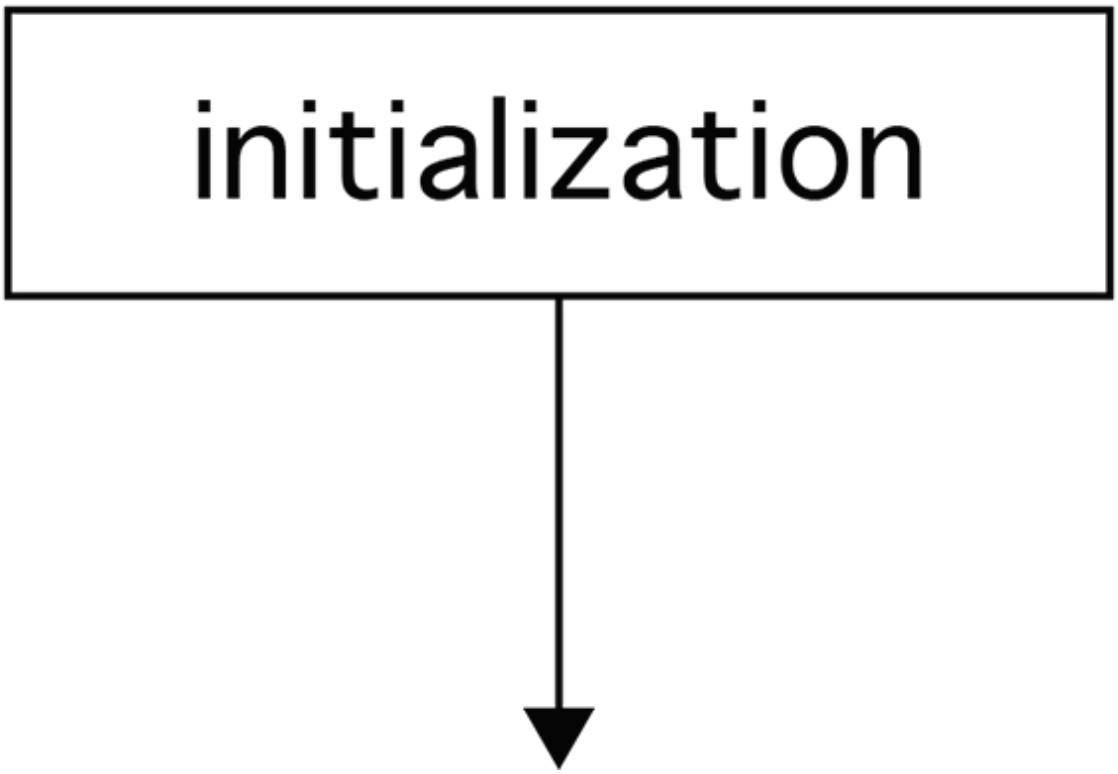


FIGURE 8-15: The initialization is the first step.

How many times will the `for` loop happen? This depends on the next part of your `for` loop: the test, shown in Figure 8-16. If the condition in the test is true, then the statements inside the curly braces will get executed. Once the test is no longer true, the `for` loop will end. We will talk more about the different types of conditions you'll create for the test in just a moment.

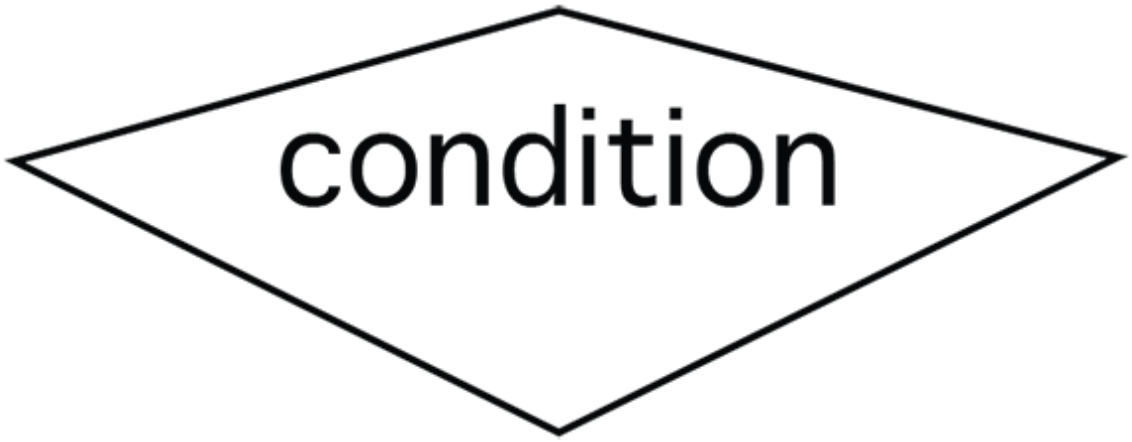


FIGURE 8-16: The condition is tested.

If the test evaluates to be true (Figure 8-17), the statements/instructions get executed. Then the value is iterated. This often means that you increase the count of your variable by one, but you can also alter the variable in a number of other ways to continue the `for` loop. Once you have iterated your variable, the `for` loop returns to the test. If the test continues to be true, the statements inside the curly brackets get executed again, and the value is iterated.

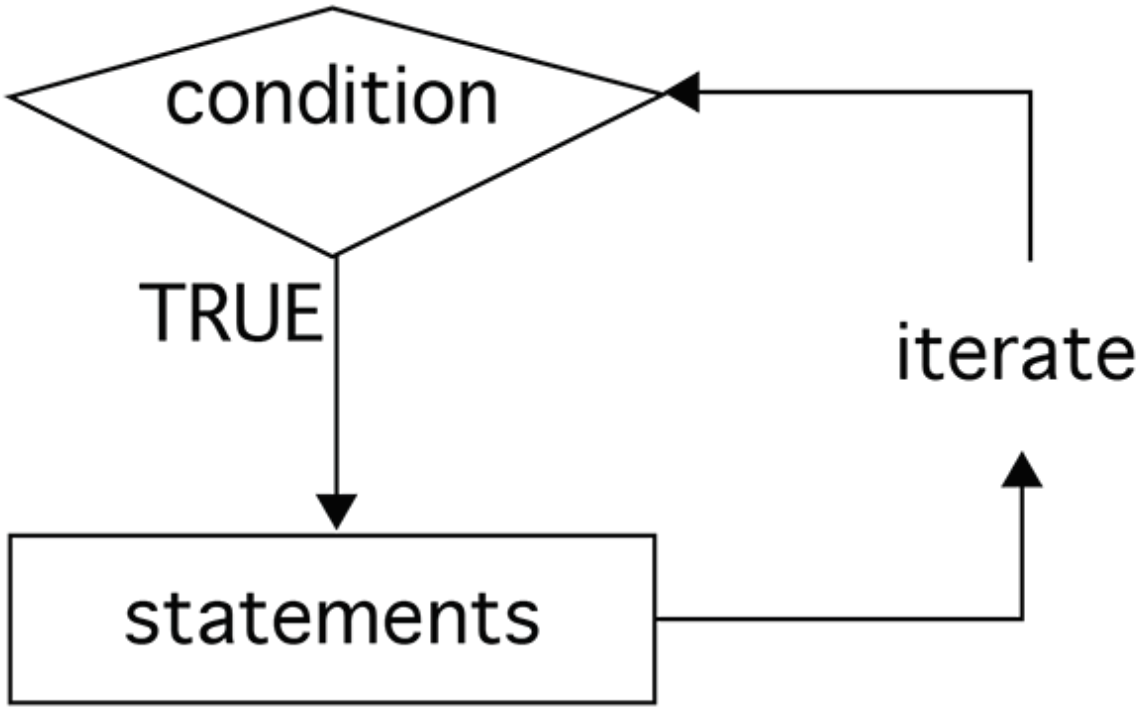


FIGURE 8-17: If the condition is true, execute statements, then iterate.

It is only when the test is false, shown in Figure 8-18, that the `for` loop ends.

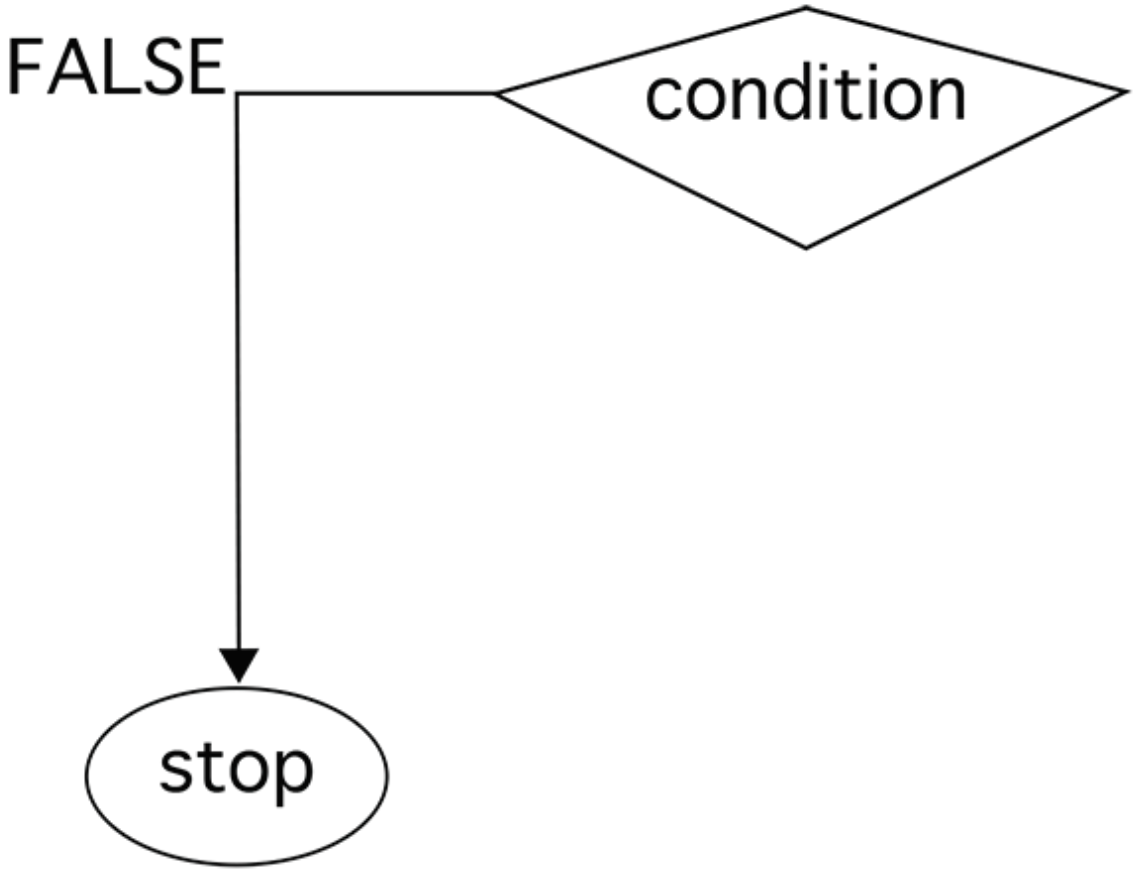


FIGURE 8-18: The `for` loop ends when the test evaluates to false.

Let's look at the cycle again with the code from our example (Figure 8-19).

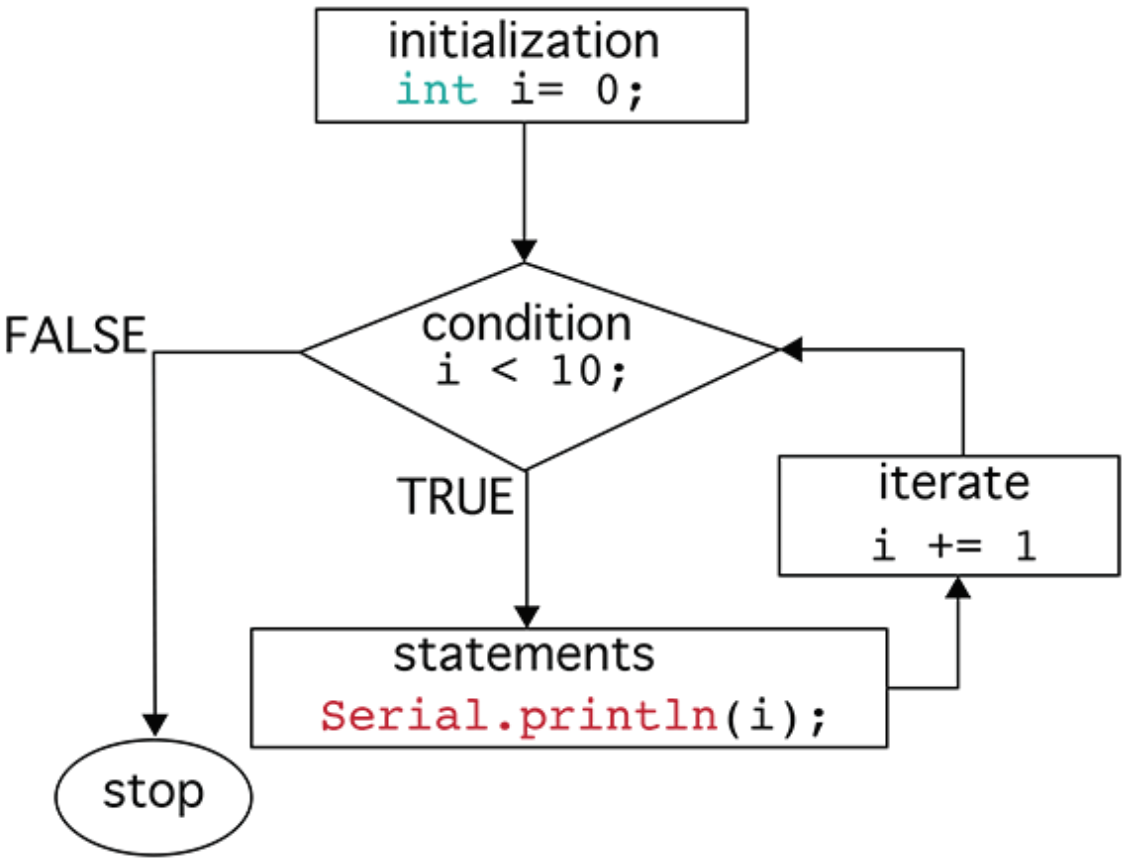


FIGURE 8-19: The `for` loop flowchart with code

Before we move on, let's look more closely at the condition, or test, section of the `for` loop, which requires discussing the idea of an *operator*.

OPERATORS

An *operator* is a mathematical or logical evaluation of values that are useful in evaluating the test in the `for` loop. In basic arithmetic, addition, subtraction, multiplication, and division are all examples of operators. There are a few different types of operators.

COMPARISON OPERATORS

Let's take a closer look at the test. You see the variable `i`, then the symbol `<` followed by `10`. What does this mean? In English, it means *is the variable i less than the integer 10*? You know that the variable `i` was set to 0 in the initialization of the `for` loop. The symbol `<` stands for "is less than"; it checks to see how the value of `i` compares to the value of 10. In this context, it is called a *comparison operator*. Comparison operators are used in logical statements, like the test in a `for` loop, or in a conditional statement, to determine whether a statement is true or false.

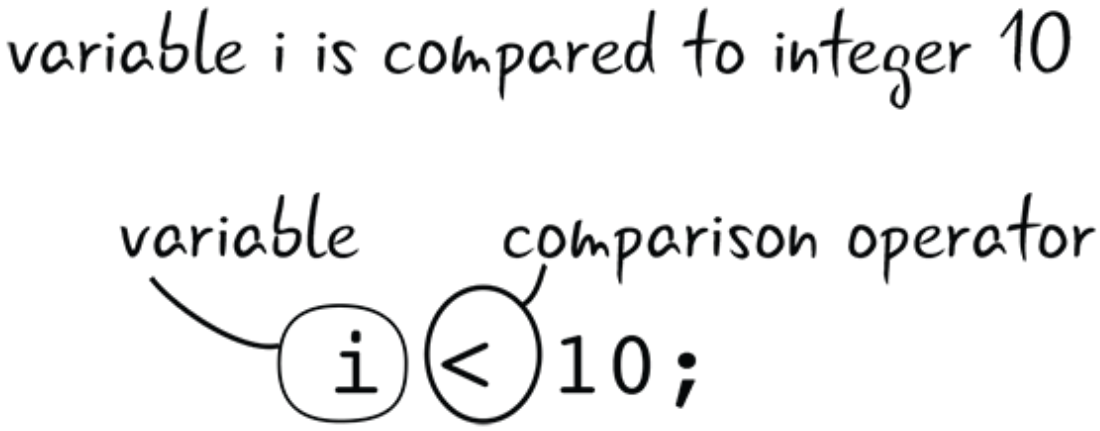


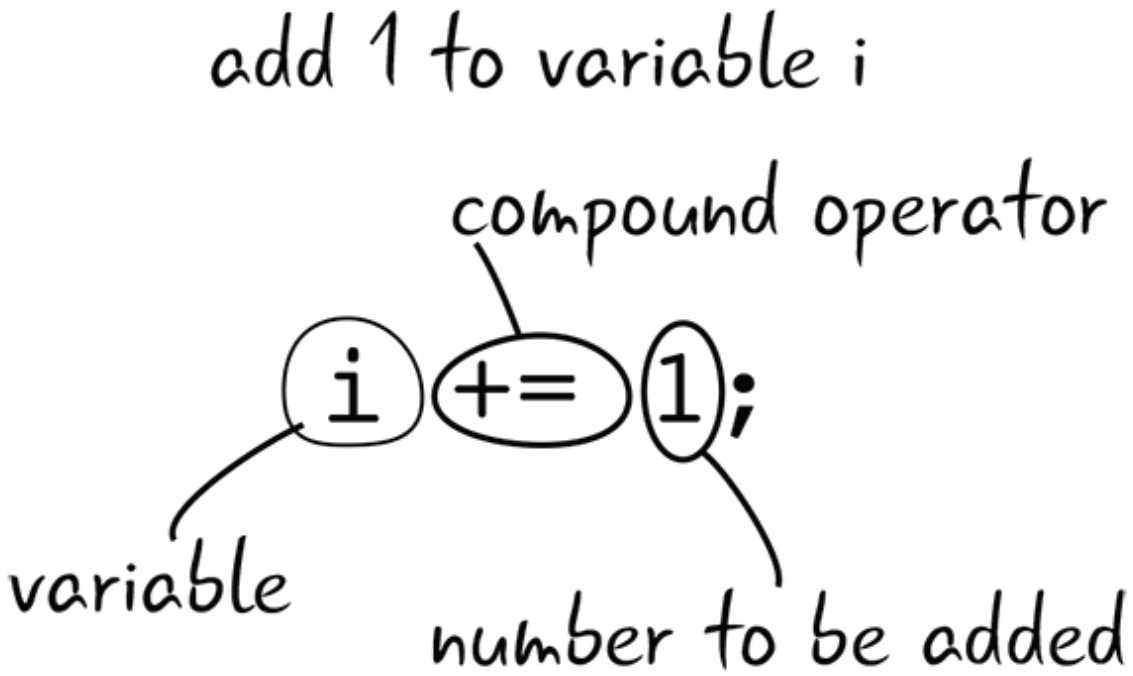
Table 8-1 shows the commonly used comparison operators in the Arduino language.

TABLE 8-1: Logical comparison operators

COMPARISON OPERATOR	WHAT IT MEANS	EXAMPLE	WHAT THE EXAMPLE MEANS
`>`	Greater than	`x > 0`	`x` is greater than 0
`<`	Less than	`x < 10`	`x` is less than 10
`>=`	Greater than or equal to	`x >= 0`	`x` is greater than or equal to 0
`<=`	Less than or equal to	`x <= 10`	`x` is less than or equal to 10
`==`	Is equal to	`x == 10`	`x` is equal to 10
`!=`	Is not equal to	`x != 10`	`x` is not equal to 10

COMPOUND OPERATORS

While we're on the topic of operators, you'll notice that a different type of operator is used in the iterator section of our `for` loop. The variable `i` is followed by `+=`, followed by 1. In English, this means you are adding 1 to the variable value. The symbols `+=` indicate that you want to add whatever is on the right side to the variable on the left side. In our example, it means add 1 to the variable `i`.



This type of operator is called a *compound operator.* Compound operators perform a mathematical operation of some kind. Table 8-2 lists commonly used compound operators in the Arduino language. In each of the examples in Table 8-2, `x` initially is set to 10.

TABLE 8-2: Results of using a compound operator when `x` initially equals 10

COMPOUND OPERATOR	WHAT IT MEANS	EXAMPLE	WHAT THE EXAMPLE MEANS
`++`	Add 1	`x++`	`x` now equals 11
`--`	Subtract 1	`x--`	`x` now equals 9
`+=`	Add value on right to value on left	`x += 2`	`x` now equals 12
`-=`	Subtract value on right from value on left	`x -= 2`	`x` now equals 8
`*=`	Multiply value on left by value on right	`x *= 5`	`x` now equals 50
`/=`	Divide value on left by value on right	`x /= 2`	`x` now equals 5

THE *FOR* LOOP IN THE SKETCH

So how can you employ `for` loops to help you move your servo? Let's take a look at the first `for` loop in our code. Breaking it down, in the initialization you set the `pos` variable to 0. The condition checks to see if the `pos` variable is less 180, and if so, a 1 is added (iterated) to `pos`. You didn't have to use `int` here to indicate the type of `pos`, because `pos` was declared in the initialization section.

```
for (pos = 0; pos <= 180; pos += 1) //goes from 0 degrees to 180 degrees
{                                   // in steps of 1 degree
    myservo.write(pos);     //tell servo to go to position in variable 'pos'
    delay(15);              //waits 15ms for the servo to reach the position
}
```

What instructions are executed each time through the `for` loop? As long as the value of `i` is less than 180, the Arduino will write the value of `pos` to the motor, which will move the servo motor to some position between 0 and 180 degrees. After the Arduino has written this position, there is a delay of 15 milliseconds.

Since the `for` loop continues for every value between 0 and 180, the servo motor will move from 0 degrees until 180 degrees in the first `for` loop. This will take a few seconds—there is a very short pause between each movement—but the motion overall will look relatively smooth. If you remove or adjust the length of the delay, the smoothness of the movement will change.

Why is the `for` loop counting between 0 and 180? Because that represents the range of movement that the shaft in your standard servo motor can move. Think of this as 0 to 180 degrees.

The second `for` loop in this sketch functions in much the same way. Instead of starting at 0, it starts with the `pos` variable equal to 180. What's 180? The second loop needs to start with the last position of the first loop, which is also the end position of the servo.

```
for (pos = 180; pos >= 0; pos -= 1) //goes from 180 degrees to 0 degrees
{
    myservo.write(pos);     //tell servo to go to position in variable 'pos'
    delay(15);              //waits 15ms for the servo to reach the position
}
```

Besides having a different starting point, this second `for` loop also decreases by one through each pass of the `for` loop. That way, the servo motor starts at 180 degrees and slowly rotates back to 0 degrees.

Once the second `for` loop has finished, the full Arduino `loop()` function has also finished. The Arduino will then return to the beginning of the `loop()` and repeat the steps, as you have seen with the other `loop()` functions.

QUESTIONS?

Q: `for` loops are used in other programming languages, right?

A: Yes, `for` loops are commonly used in many different programming languages. They are often used when something needs to happen a certain number of times.

Q: Are there other types of loops besides the `for` loop in the Arduino programming language?

A: Yes, there are `while` loops and `do` loops. Read more about them here: arduino.cc/en/Reference/While and arduino.cc/en/Reference/DoWhile.

THINK ABOUT IT...

Now that you know how a servo motor operates, think of some uses for them. What types of devices have you seen that use servo motors? What are projects you would like to build that would require this kind of movement?

ADD INTERACTIVITY: TURN THE FLAG

You now have a basic understanding of how the servo motor functions with your Arduino code, so let's try making your servo

circuit interactive. Rather than the Arduino moving the servo at a steady pace continually, this next sketch uses information from a potentiometer to position the shaft of the servo motor. As you turn the knob, the shaft of the servo motor will move.

You will use the Knob sketch that is also included in the servo motor examples in the IDE. Before you upload the sketch, though, let's adjust your circuit by adding the potentiometer.

ADDING A POTENTIOMETER STEP BY STEP

Adding a potentiometer to the circuit allows you to control how the flag waves and to set it to a precise position. If you completed the first circuit with the servo, you can leave the servo control wire attached to Pin 9 on the Arduino, the power line attached to the power bus, and the ground wire attached to the ground bus. You will be adding only the potentiometer to the circuit.

You'll need these parts:

1 10 K potentiometer

Jumper wires

Figure 8-20 shows the finished circuit with the schematic.

> **Tip**
>
> As always, make sure you have unplugged the Arduino before you make any changes to the circuit.

Place the potentiometer in the breadboard, as shown in Figure 8-21.

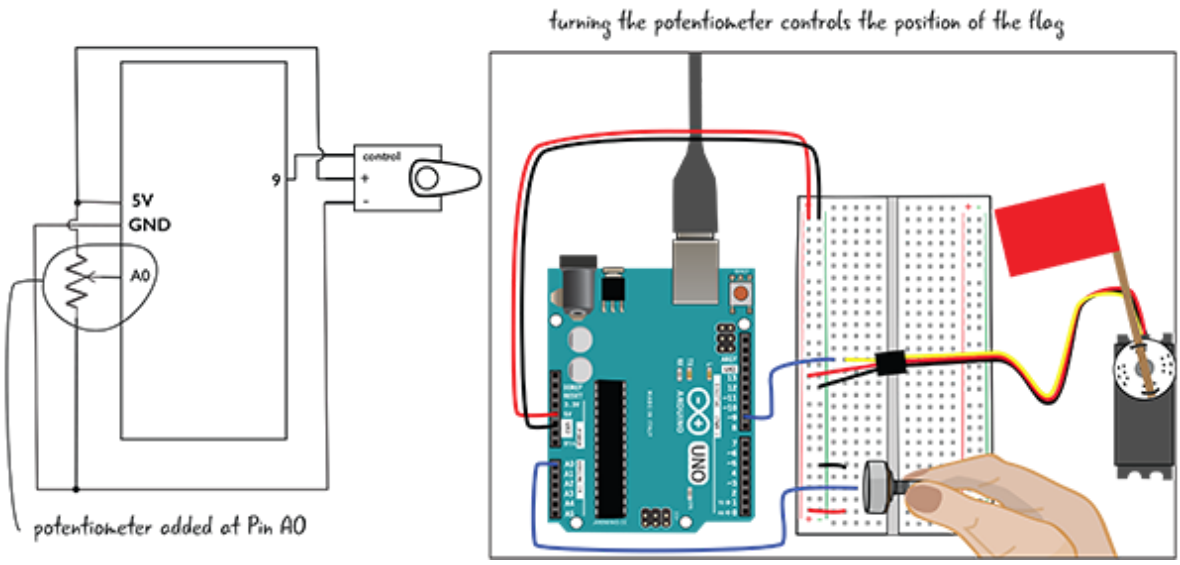


FIGURE 8-20: Circuit with the potentiometer added

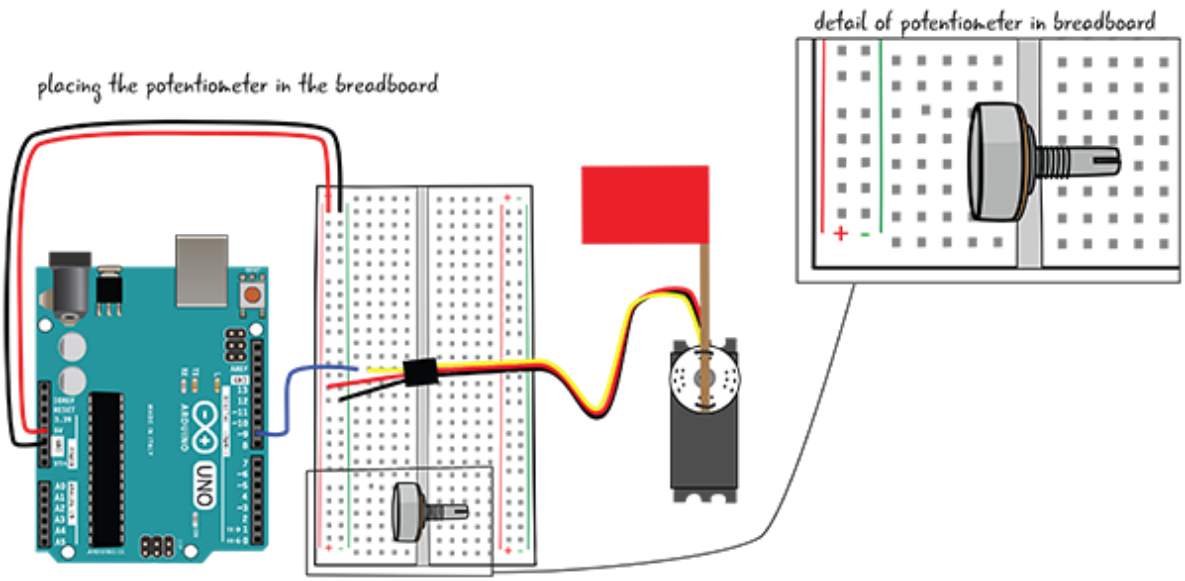


FIGURE 8-21: Adding the potentiometer to the breadboard

Connect one end of the potentiometer to the ground bus with a jumper. Connect the other end of the potentiometer to the power bus with a jumper. Connect the middle pin of the potentiometer to Pin A0, one of the analog input pins (Figure 8-22).

Now that you've wired your circuit, it's time to open the next sketch in the Arduino IDE. This sketch is located under File >

Examples > Servo > Knob. Once you have opened it, save it as LEA8_Knob. Hook up your computer to your Arduino. Click Verify to check the code, and then click Upload to load it to your Arduino.

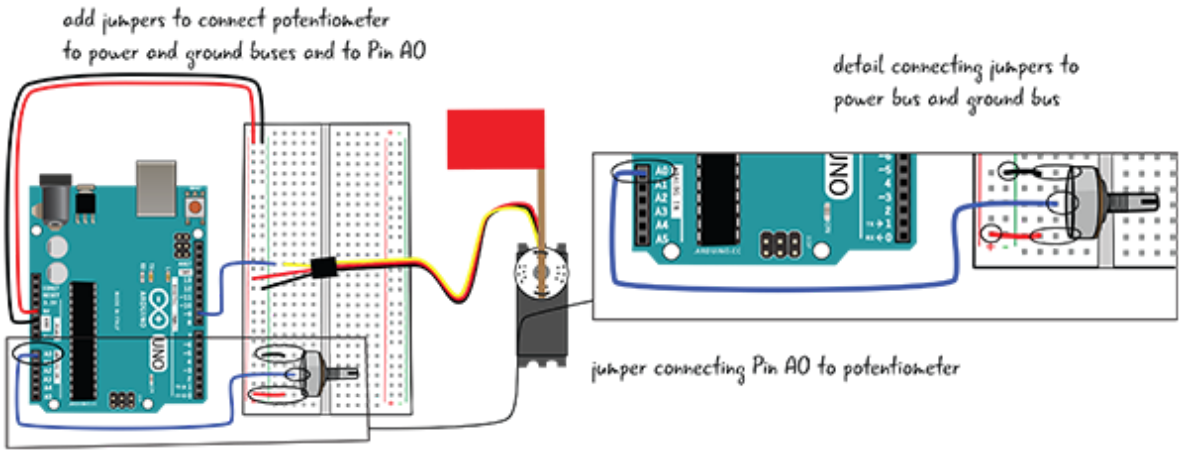
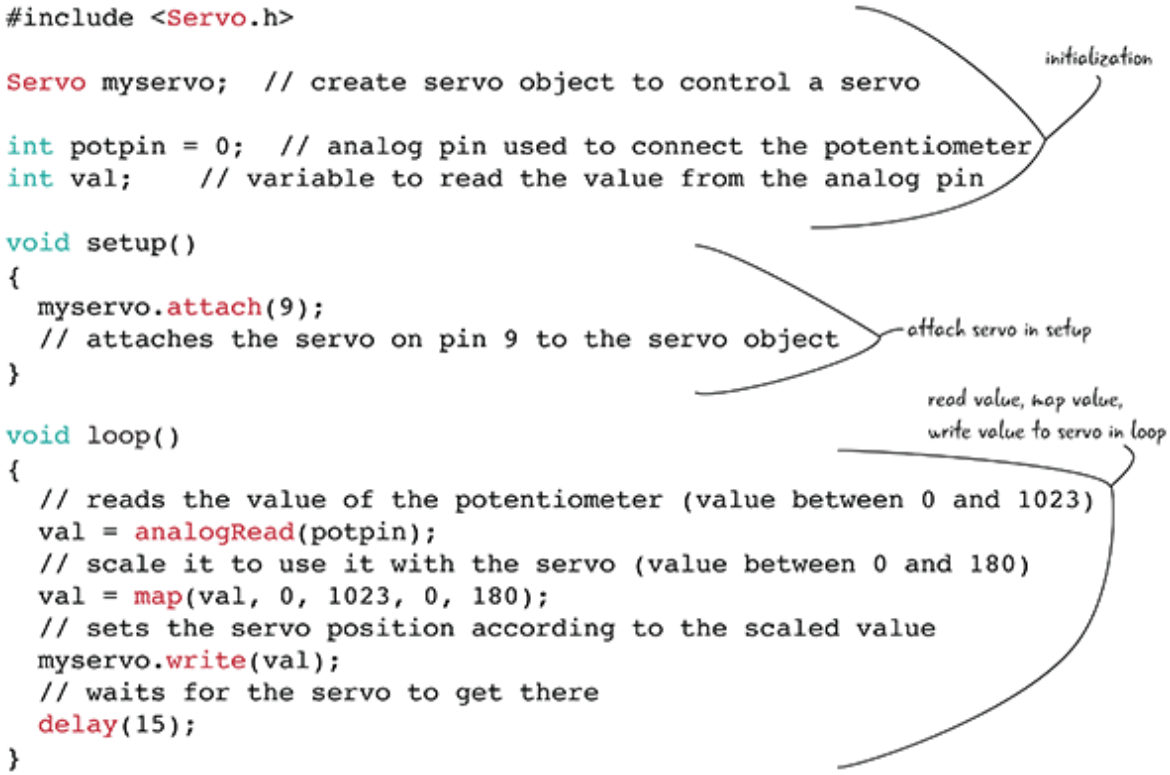


FIGURE 8-22: Adding jumpers to the potentiometer

Now when you turn the potentiometer, your flag should turn, too.

HOW DOES THE SKETCH CHANGE WHEN WE ARE USING A POTENTIOMETER?

Let's take a quick look at the sketch. It is similar to the LEA8_Sweep sketch—with a couple of important differences that we will look at closely.




```
#include <Servo.h>

Servo myservo;  // create servo object to control a servo

int potpin = 0;  // analog pin used to connect the potentiometer
int val;    // variable to read the value from the analog pin

void setup()
{
  myservo.attach(9);
  // attaches the servo on pin 9 to the servo object
}

void loop()
{
  // reads the value of the potentiometer (value between 0 and 1023)
  val = analogRead(potpin);
  // scale it to use it with the servo (value between 0 and 180)
  val = map(val, 0, 1023, 0, 180);
  // sets the servo position according to the scaled value
  myservo.write(val);
  // waits for the servo to get there
  delay(15);
}
```

LEA8_KNOB EXPLAINED

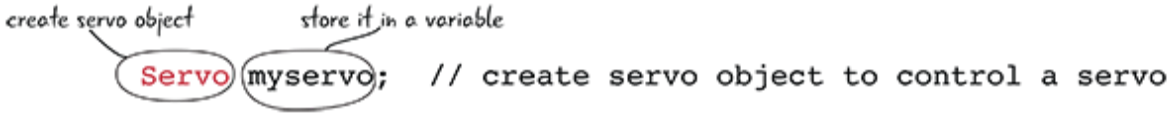
Just like with LEA8_Sweep, this sketch controls the position of the horn attached to the shaft of the servo motor, depending on what value the Arduino sends to it. However, this time you have control over how much it turns, since as you turn the potentiometer, you change the value that the Arduino receives and sends to the servo.

INITIALIZATION

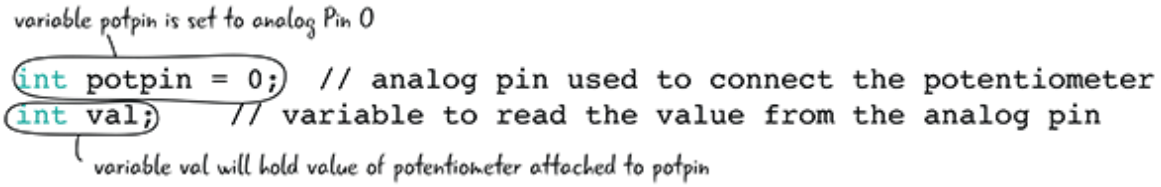
In this servo sketch, as with the previous one, the first thing you see in the initialization section is the include statement that loads the Servo library. As you've seen, libraries extend the abilities of the Arduino to perform specific functions or interact with some types of technology in a streamlined way so that you can write simplified code.

```
#include <Servo.h>
```

Looking at the next line of the initialization section, you see that, as in LEA8_Sweep, you're creating a new servo object named `myservo`. This object will be able to access the functions of the Servo library to communicate with the servo motor.

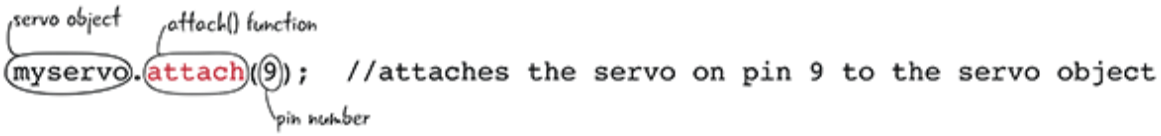


The initialization section also contains a variable for the analog pin to which your potentiometer is attached. We covered this in Chapter 7; by connecting the potentiometer to Analog Pin 0 you are able to take readings between 0 and 1023 instead of the `HIGH` or `LOW` reading you get from a digital pin. Finally, the last line in the initialization sketch creates a variable called `val`, which you will later use to store the value coming in from the potentiometer and send it out to the servo.



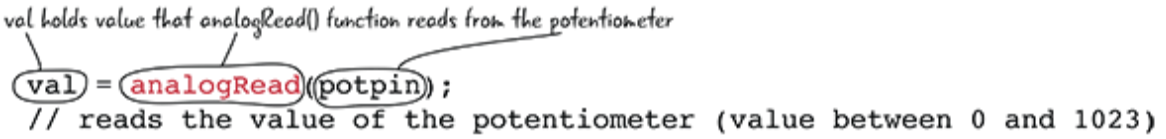
THE CODE IN *SETUP()*

The `setup()` section includes only a single line for this sketch. Again, you use `attach()` to connect the Arduino to the servo object that you named `myservo` to a pin on your Arduino. That way, whenever you refer to `myservo`, you are referencing to a particular pin much in the way that you have seen with `digitalWrite()` and a pin name. In this case, your servo should be wired so that it is attached to Pin 9.

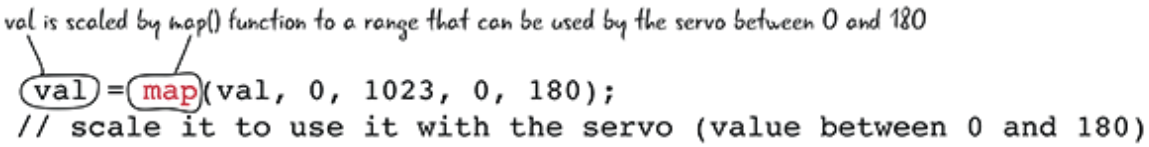


THE CODE IN *LOOP()*

The `loop()` code section looks similar to what you did in Chapter 7. The first step is to use the `analogRead()` function to read the value from the potentiometer on Pin A0 and store it in the variable `val`. This will set `val` to a value between 0 and 1023, which as you know is the range of possible values from an analog pin.



Next, you use the `map()` function to adjust the value from your potentiometer reading to match up with the degrees of motion for your servo motor. Since your servo motor is able to move 180 degrees, you will scale the value from 0 to 180. That way, when you send the value to the servo motor, it will already be in a value given in degrees. This new scaled value is then again saved in your `val` variable.



The next step is to write out your scaled `val` variable to the servo attached to Pin 9 using the `write()` function of the servo object. It is worth noting again here that it will not move the servo an additional `val` number of degrees, but that it will move to the `val` number of degrees from 0. For example, if `val` is equal to 90, it will always move the servo shaft to the midpoint.

```
// sets the servo position according to the scaled value
myservo.write(val);
```

The last line in the `loop()` code delays the Arduino program for 15 milliseconds. This very short delay time will let the servo move to the correct position since the movement is not instantaneous. It will also give the Arduino slightly more time between potentiometer readings to ensure a more accurate reading overall.

```
// waits for the servo to get there
 delay(15);
```

TWO FLAGS WAVING: ADD A SECOND SERVO MOTOR

Let's add another servo motor to the circuit. You'll use the information from the potentiometer to set the position of both servos. You're making a flag-waving signal system.

In the sketch for this project, you'll learn how to write a custom function, and you'll also learn more about using logic in conditional statements.

If you built the circuit for the LEA8_Knob sketch, this circuit will be almost the same—the only addition is the second servo motor.

You'll need the following:

Servo motor

Jumpers

Coffee stirrer or cardboard strip

Colored paper

Tape

Figure 8-23 shows the schematic for the circuit, as well as a drawing of the completed project.

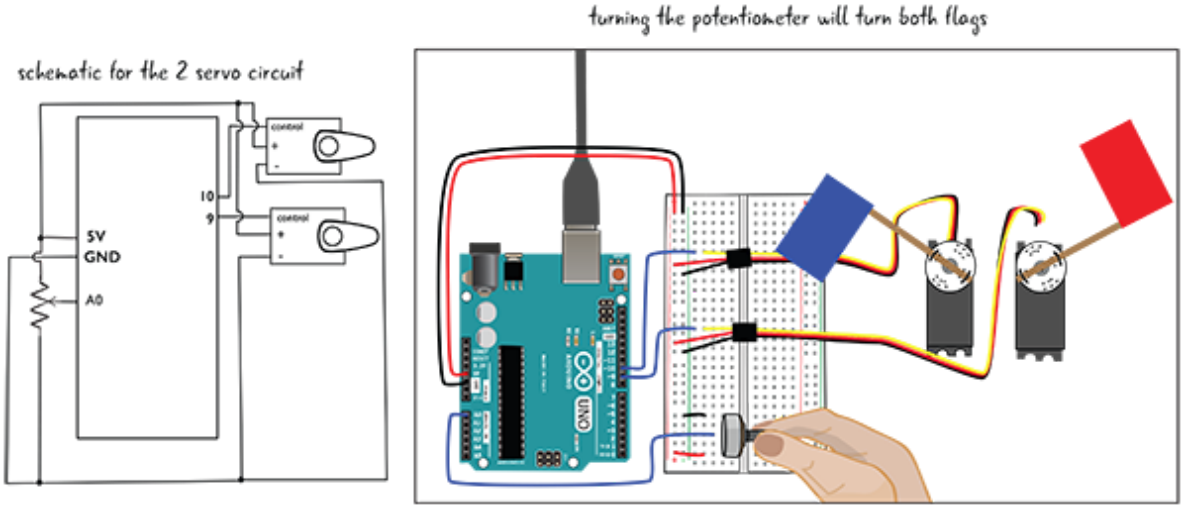


FIGURE 8-23: Two-servo circuit and schematic

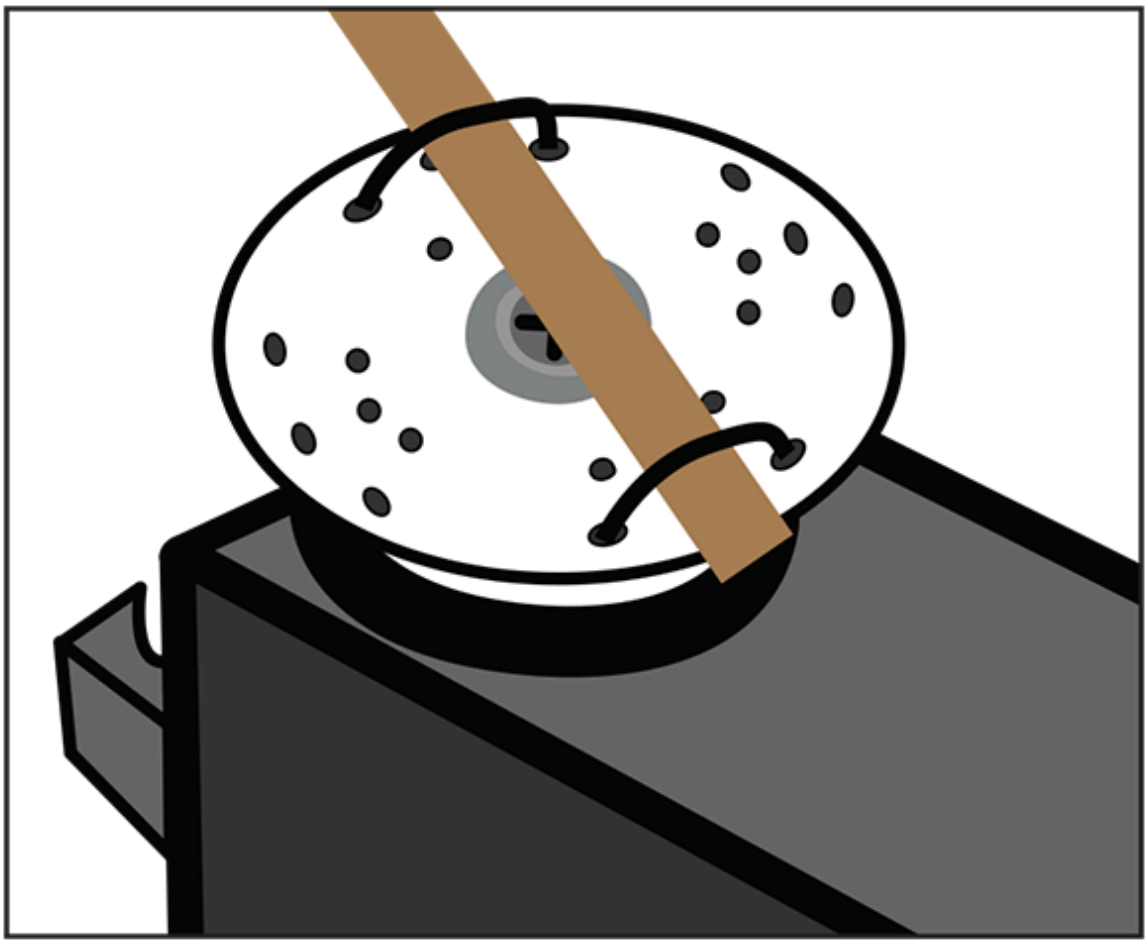


FIGURE 8-24: Attach the flag to the servo horn.

First attach the wooden stirrer with the paper flag to the servo horn, as you did with the first servo motor (Figure 8-24).

Attach the jumpers to the servo connector (Figure 8-25), matching the colors for power and ground, and attach the control wire.

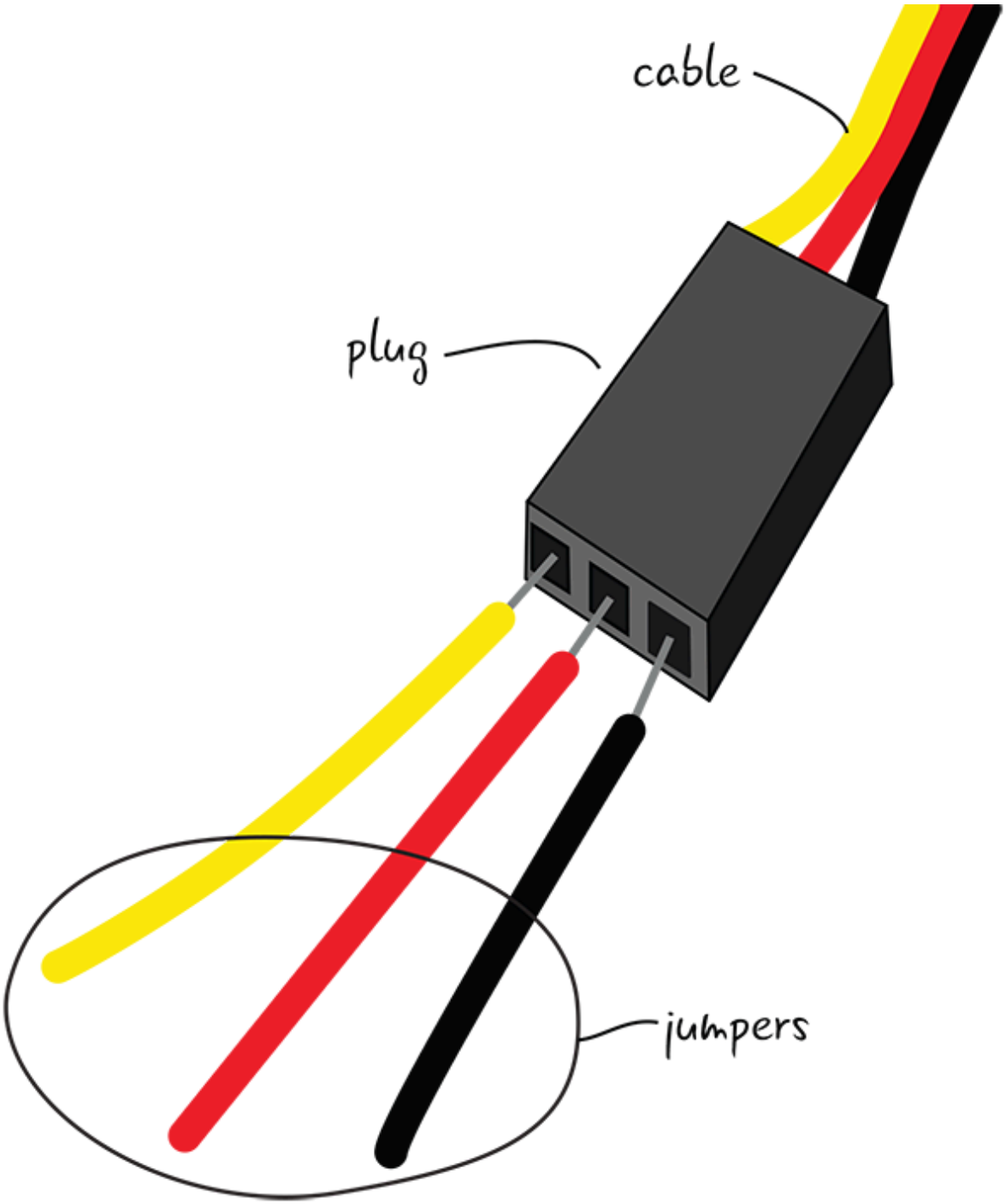


FIGURE 8-25: Attach jumpers to the servo connector.

Now attach the jumpers to the breadboard, as shown in Figure 8-26. As you did before, connect the jumper connected to the power cable to the power bus and the jumper connected to the ground

cable to the ground bus. The control wire connects to a row of tie points. Finally, connect Pin 10 to the same row of tie points as the jumper from the control cable.

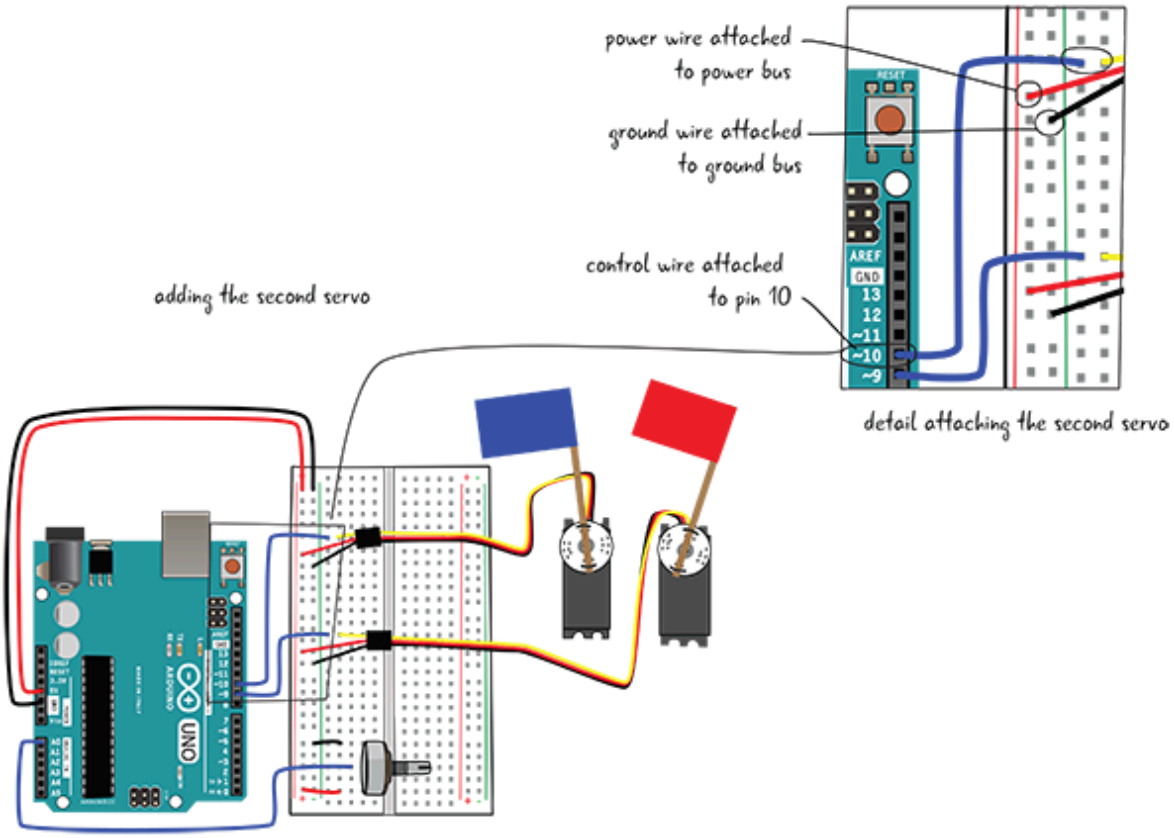


FIGURE 8-26: Attaching the second servo

Before you attach the Arduino to your computer, you must make some adjustments to the code. Let's look at the sketch.

LEA8_2_SERVOS, FIRST LOOK

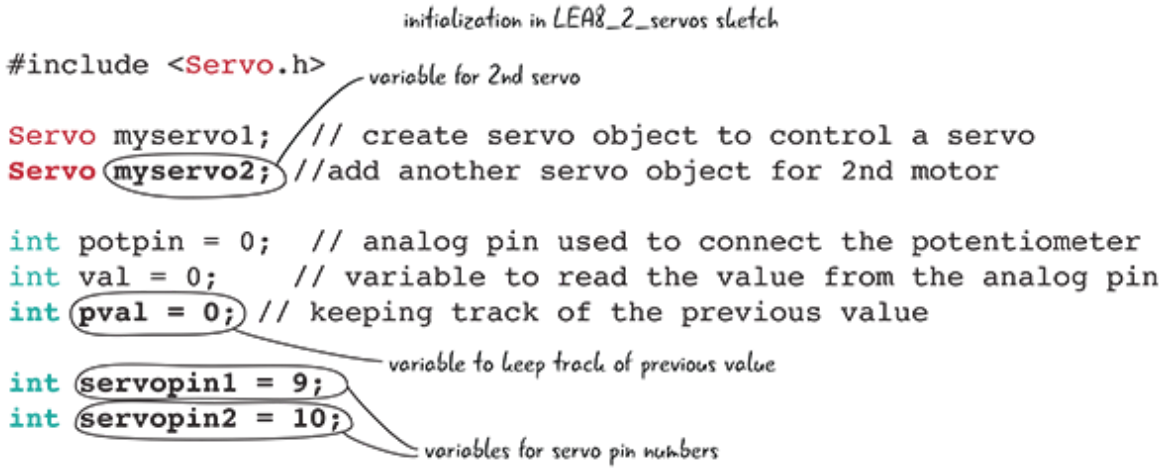
Save LEA8_Knob as LEA8_2_servos. You will be adjusting this code. In the LEA8_2_servos sketch, the initialization section is similar to your other sketches, as is `setup()`, but you'll notice something new in `loop()`, which we will explain. Figure 8-27 is a first look at the code.

FIGURE 8-27: The LEA8_2_servos sketch annotated

Some of the comments have been moved in the code breakdowns for legibility.

INITIALIZATION

As we've said, the initialization section is quite similar to that section in LEA8_Knob. There are a couple of additions: you're including a variable for the second servo object, and you're storing the pin numbers that the servos are attached to in variables. You're also adding a variable (`pval`) to store the previous value.



THE *SETUP()* FUNCTION IN LEA8_2_SERVOS

The `setup()` function has a couple of changes from the LEA8_Knob sketch. You're attaching the servo objects to the pins, using the variables `servopin1` and `servopin2`. Then you're using the `write()` function of the Servo library to move the first servo to midpoint, followed by moving the second servo to midpoint using the `write()` function again.

code in setup()

```
void setup(){
  myservo1.attach(servopin1);
    // attaches the servo on pin 9 to the servo object
  myservo2.attach(servopin2);
    //attach 2nd servo
  myservo1.write(90);
    //move 1st servo to midpoint
  myservo2.write(90);
    //move 2nd servo to midpoint
}
```

attaches servo on pin 9 to servo object

attaches servo on pin 10 to servo object

set both servos to midpoint

OVERVIEW OF THE *LOOP()* CODE

In the `loop()` code, the first line saves the previous value read from the potentiometer into the variable `pval`. You then read the current value on the potentiometer and map it. Finally, you use a conditional statement to check to see if `val` is not equal to `pval`—in other words, check to see if there has been a change in the pin reading. If there has been a change, you call the `turnServos()` function.

You'll notice a few things you haven't seen before in this code. Let's break it down.

loop() code

```
void loop(){
  pval = val;
    //set previous value from potentiometer reading to current reading
  val = analogRead(potpin);
    //check value from potentiometer on pin A0
  val = map(val, 0, 1023, 0, 180);
    //map value to range used by servo
  if (val!=pval){
    // if there has been a change, call turnServos function
    turnServos();
  }
}
```

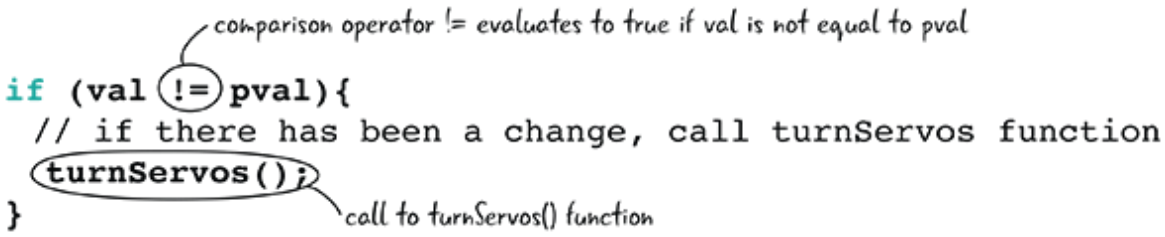
store previous value

check if value of potentiometer has changed

if changed, call function

We use a conditional statement to compare the previous value read from the potentiometer to the new value read. This conditional uses

a *comparison operator*: the symbols `!=`. This operator is used to compare two values, and if one value is *not* equal to the other, it evaluates to true. (Table 8-1 covered these comparison operators and what each means.)



If there has been a change, the `turnServos()` function is called. `turnServos()` is a *custom function*, the topic of our next section.

CREATING A CUSTOM FUNCTION

We've introduced another new code concept in the updated servo code: custom functions. Why would you want to make your own functions? First let's quickly review what a function is.

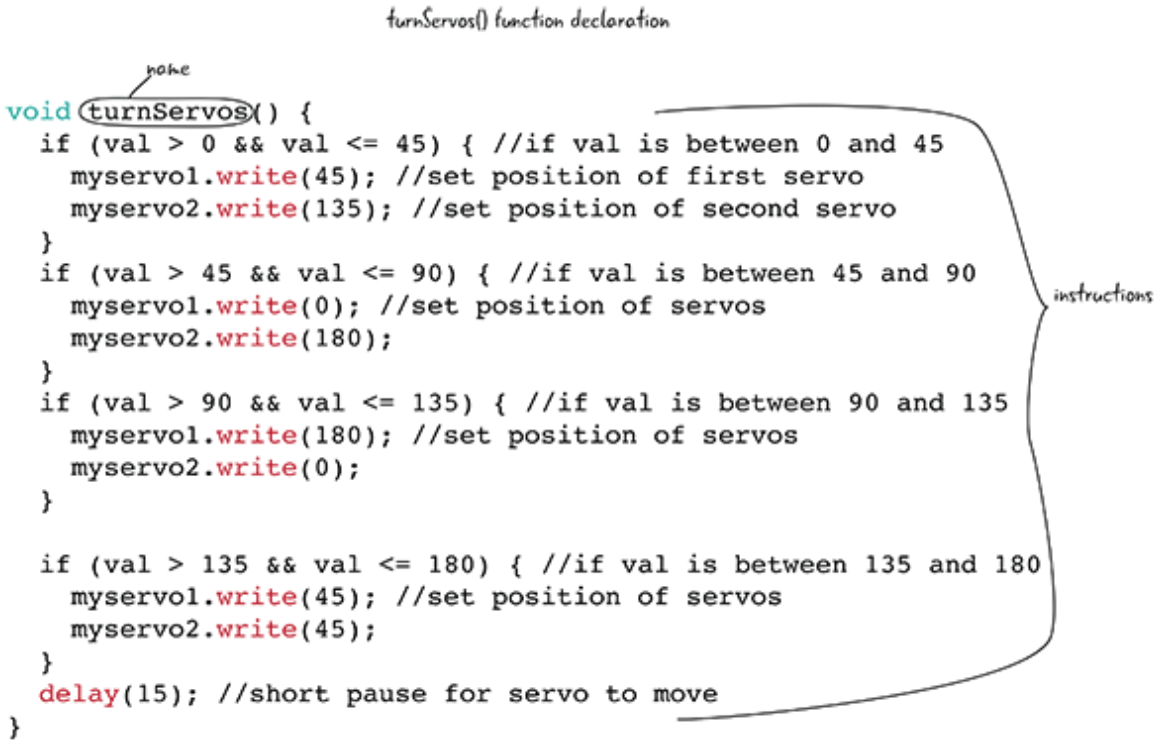
> **Note**
>
> A *function* is a block of code that performs a specific action or series of actions that can be used over and over.

You have used many Arduino functions: `delay()`, `digitalWrite()`, and `analogRead()` are all functions used to perform some specific task with the Uno. You have written most of your code inside the `setup()` and `loop()` functions for your sketches.

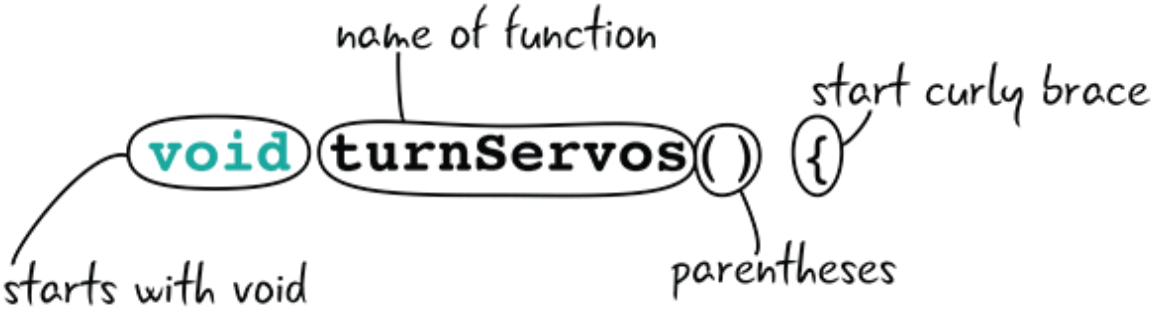
What is the advantage to writing your own functions? You can group together actions outside of your `loop()` code and call this new function only when you want those actions to happen. It makes your code more legible and easier to understand.

What does the line of code `turnServos();` do? It's the call to the custom function you wrote. As you've seen, it will be called if there has been a change to the variable `val`—in other words, if someone has turned the potentiometer. `turnServos()` will only happen when the value of the potentiometer has been changed and not every time the Arduino goes through the `loop()` function.

What does your `turnServos()` function look like? First, it starts with `void`. This is followed by `turnServos`, which is the name of the function, followed by parentheses and an opening curly brace. The instructions to be executed when `turnServos()` gets called follow, and the last line contains only a closing curly brace.



Creating a custom function is called *declaring a function*, and it follows a few rules in the Arduino programming language.



Note

There are a couple of simple rules for naming functions. Functions must start with a letter and can't be named the same as an Arduino reserved word. And it is best to make the name of your function clearly indicate what exactly it is going to do.

Note

Why does the function start with `void`? It has to do with how some functions operate; in some functions you might see `int` or `string` there, for example. It's beyond the scope of this book to explore this concept, but here's an explanation from the Arduino site: arduino.cc/en/Reference/FunctionDeclaration.

Next you see a set of parentheses. Some functions have parameters or information that will be passed into the function as arguments when the function is called. These are placed inside the parentheses. Since `turnServos()` doesn't have any parameters, the parentheses are empty. The parentheses are followed by an opening

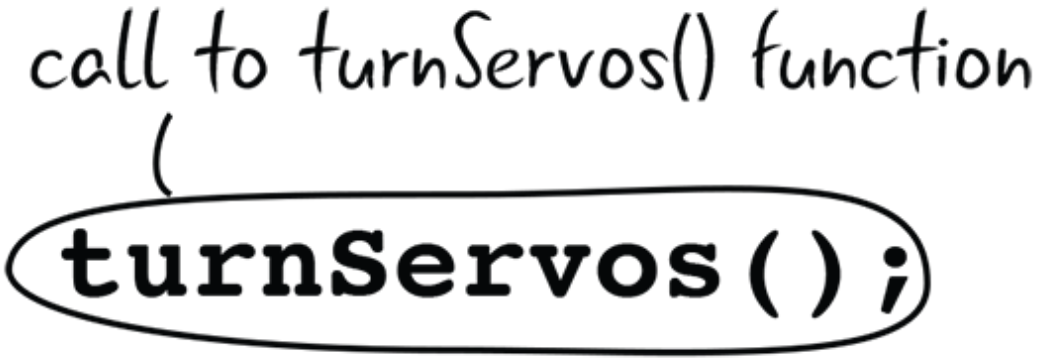
curly brace, the punctuation you have used before to mark out a block of code.

This discussion about function declaration must seem familiar, because you have seen some of these conventions used before. Where? In `setup()` and `loop()`! The difference is that now you are creating and naming the function yourself.

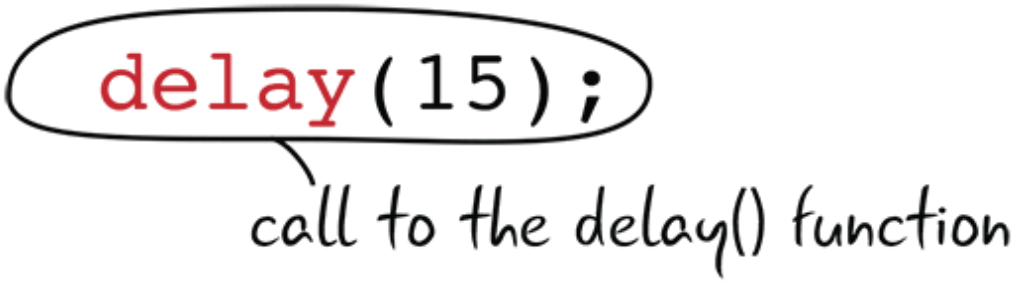
You can write your own functions at any time, and your functions can incorporate any Arduino-compatible code.

CALLING A CUSTOM FUNCTION

When you want to invoke a function, you *call* that function. The call to your custom function is inside `loop()`. The call is simply the name of the function followed by parentheses and a semicolon.



You've been making calls to the built-in Arduino functions since you uploaded your first sketch. The functions you've used have usually had parameters, so you passed in arguments inside the parentheses. When you called the `delay()` function, for example, you passed in the amount of time in milliseconds that you wanted the delay to last.

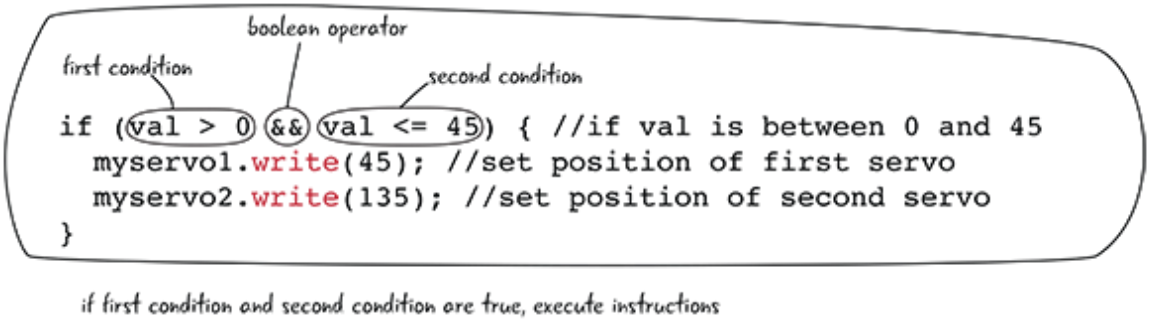


Custom functions can be quite powerful by letting you extend the functionality of your sketches. Our discussion here is limited—it is meant to give you an introduction to the concept and the general rules for creating them. You will undoubtedly explore more on your own.

INSIDE *TURNSERVOS()*

You know that `turnServos()` is going to turn your motors, but how? It is positioning both flags in a pattern based on how far the potentiometer is turned; sometimes they are opposite each other, and sometimes they are parallel. The code consists of a series of conditional statements. Let's look closely at the first one.

In this conditional, it is testing for two conditions. Is `val` greater than 0 *and* is `val` also less than or equal to 45? The value of `val` must be a number between 0 and 45 for the instructions inside the `if` statement to be executed.



The symbols `&&` are an example of a *Boolean operator*. In this case, both the first and second conditions must be true in order for the

servo positions to be set.

Boolean Operators

Boolean operators allow you to make complicated evaluations when trying to decide what actions should be taken. Table 8-3 lists the Boolean operators and what they mean and includes an example for each.

TABLE 8-3: Boolean operators

BOOLEAN OPERATOR	WHAT IT MEANS	EXAMPLE	WHAT THE EXAMPLE MEANS
`&&`	logical and	`if (a>0 && b<10)`	Evaluates to true if both conditions are true
`||`	logical or	`if (a>0 || b<10)`	Evaluates to true if either condition is true
`!`	not	`if (!a)`	Evaluates to true if `a` is false

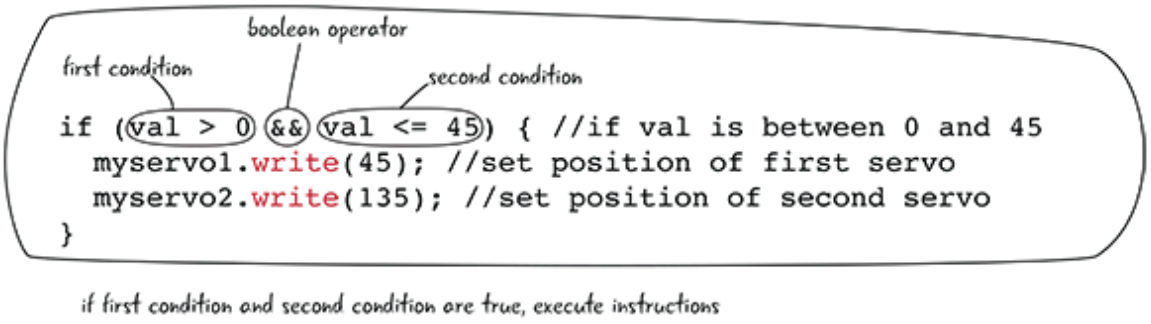
THINK ABOUT IT...

Although you use `&&` only in this sketch, can you think of how you might use the other operators to change the execution/logic of your program?

The *turnServo()* Function and Boolean Operators

Let's take a look at the first `if` statement in `turnServos()` again. What happens if `val` is a number between 0 and 45? The first servo

motor turns to a position of 45 degrees and the second servo motor turns to a position of 135 degrees. You know this because both of the servo object `write()` functions will be called and move `myservo1` and `myservo2` to the desired position.



The three other conditional statements in `turnServos()` work in a similar manner—testing the value of `val` (how far the potentiometer has been turned), and whether it is between a particular range of numbers, the Arduino will then turn each of the servo motors to their new position specified by the `turnServos()` function.

Now that you've written the sketch, make sure you've saved it (as LEA8_2_servos) if you haven't already done so. Click the Verify button to check for errors, and if it is error free, click the Upload button. Your flags will change positions as you turn the potentiometer (Figure 8-28).

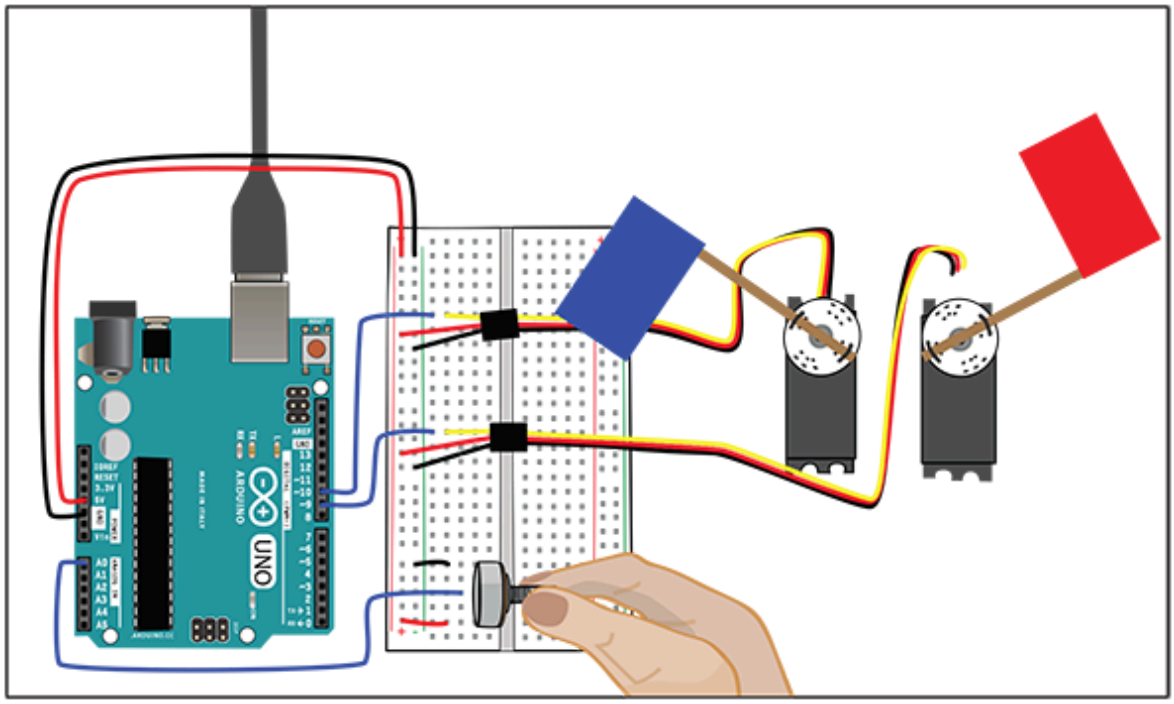


FIGURE 8-28: Waving two flags

QUESTIONS?

Q: When should I write my own custom functions?

A: If you know there is a block of code you will need often in a sketch, or if you find yourself repeating the same lines, it may help to write a custom function to shorten your code and make it easier to read. We could have used a custom function earlier in our first SOS sketch. In fact, we could have used `for` loops there as well.

Q: How do I know which Boolean or comparison operator to use?

A: As with most programming concepts covered in this book, it will be easiest to state in plain language what you are trying to accomplish in order to decide how the logic will be structured in your sketch. For example, if you want conditions to be true (button 1 is pressed and an LED is lit) to make something happen, you would use `and`, or the `&&` sign, as you did in this sketch.

Q: How many is too many conditionals?

A: We selected four key positions in this sketch in order to create a choreography of sorts for the flags. You can use as many conditionals as you need to get the kind of behavior that your project requires.

SUMMARY

Our primary focus in this chapter has been to show you how to use servo motors. Servo motors are versatile for many Arduino projects,

since a servo can easily be run automatically, as in the Sweep sketch, or can be controlled by a sensor or switch, as in the Knob sketch.

We discussed a number of important programming concepts in this chapter. You learned about libraries, and you used the Servo library to give your Arduino access to a number of servo functions that make it easier to control the servo motors.

We also showed you how to use `for` loops to set your servos to different positions. And you learned about using comparison, compound, and Boolean operators in your code. You'll find the LEA8_2_servos sketch here: github.com/arduinotogo/LEA/blob/master/LEA8_2_servos.ino.

9

BUILDING YOUR PROJECTS

Now that you've completed the projects in this book, what's next? This chapter will be a brief overview of tips for project management, a few project ideas, and a quick look at some of the other Arduino boards available and what they can do.

PROJECT MANAGEMENT

In this book, we gave you step-by-step instructions on how to work with the Arduino. How do you start your own projects? The first step should be research. Look around online; many of the vendors we've mentioned have websites that are chock-full of tutorials and project ideas. Also, browsing through sites to get an idea of inputs and outputs that are available to you should give you plenty of ideas. Here are a few sites:

makezine.com/category/technology/arduino/

learn.adafruit.com/category/learn-arduino

learn.sparkfun.com/tutorials/tags/arduino?page=all

playground.arduino.cc/Projects/Ideas

OUTLINE YOUR PROJECT

Once you have an idea for a project, try sketching or writing out the system that you are thinking of building. This can be as simple as making a list containing the components you are planning to use and the type of behavior you will need in the code. It usually helps to be able to break your project down into inputs, outputs, and code. Remember, your project will always be a system, with inputs, outputs, and code that controls behavior running on the Arduino (Figure 9-1).

BREAK IT DOWN

Breaking your project down into component parts, starting with the simplest section that you already know how to do, will help you get the work underway. Tackling each part separately, rather than facing the entire scope, makes it easier to get the job done. Also, when working initially, simplifying your idea will help you realize it; it's always possible to enhance and refine it in later versions.

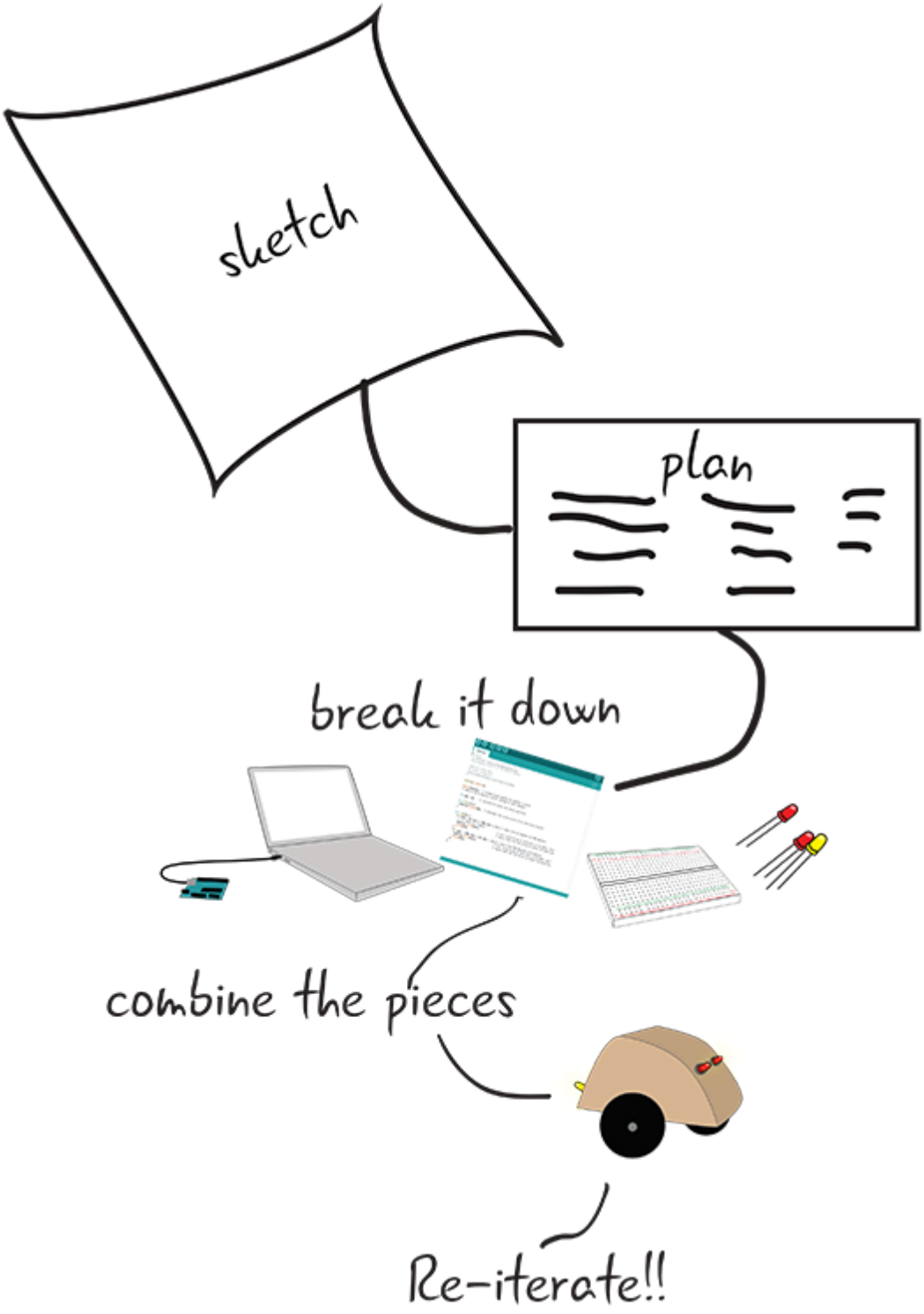


FIGURE 9-1: Planning notes can help direct your project.

As you start building your project, what do you do if something isn't working? Throughout this book we have emphasized the importance of debugging, both your code and your circuits (Figure 9-2). Be patient and apply a methodical approach to examining each element of your project. If you get an error in the code editor of the IDE, note the exact language and type it into a search engine. You will probably find that you are not the first person to have this problem. The forums on the Arduino website (forum.arduino.cc) are a great place to search for answers to problems and post questions. Arduino Stack Exchange (arduino.stackexchange.com) is another place to look.

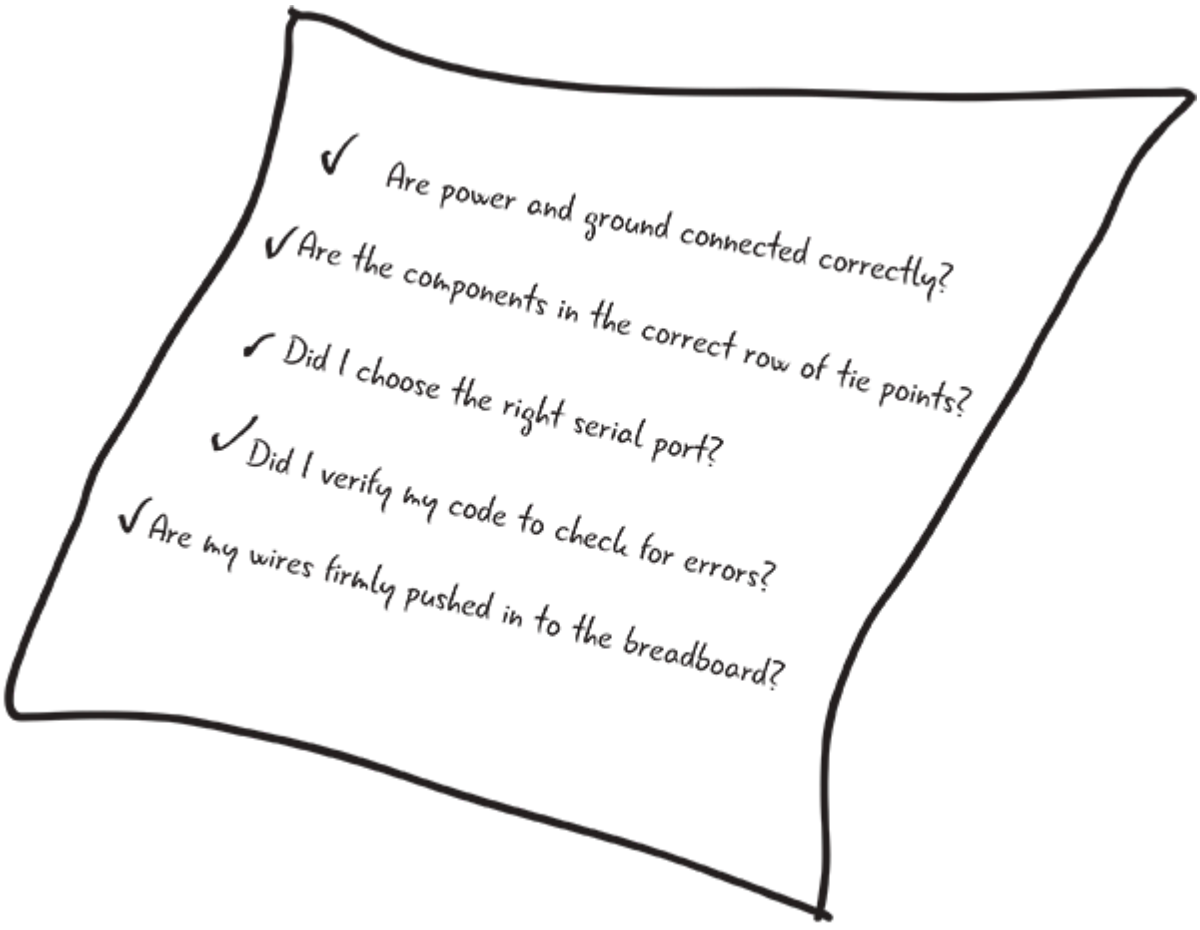


FIGURE 9-2: Avoid the frustrations of broken projects—use debugging.

USER TESTING

Once you have a working version or prototype of your project, share it with someone. Explain the project to them and have them test out your device. When building a project, it is easy to make a lot of assumptions about how someone will see it or use your project, and it can help to have an outside perspective in order to break some of your assumptions. If possible, having a wide variety of people test your project will help to make it the best possible version and to develop your idea. If you are unsure who to approach, start with friends and family (Figure 9-3).

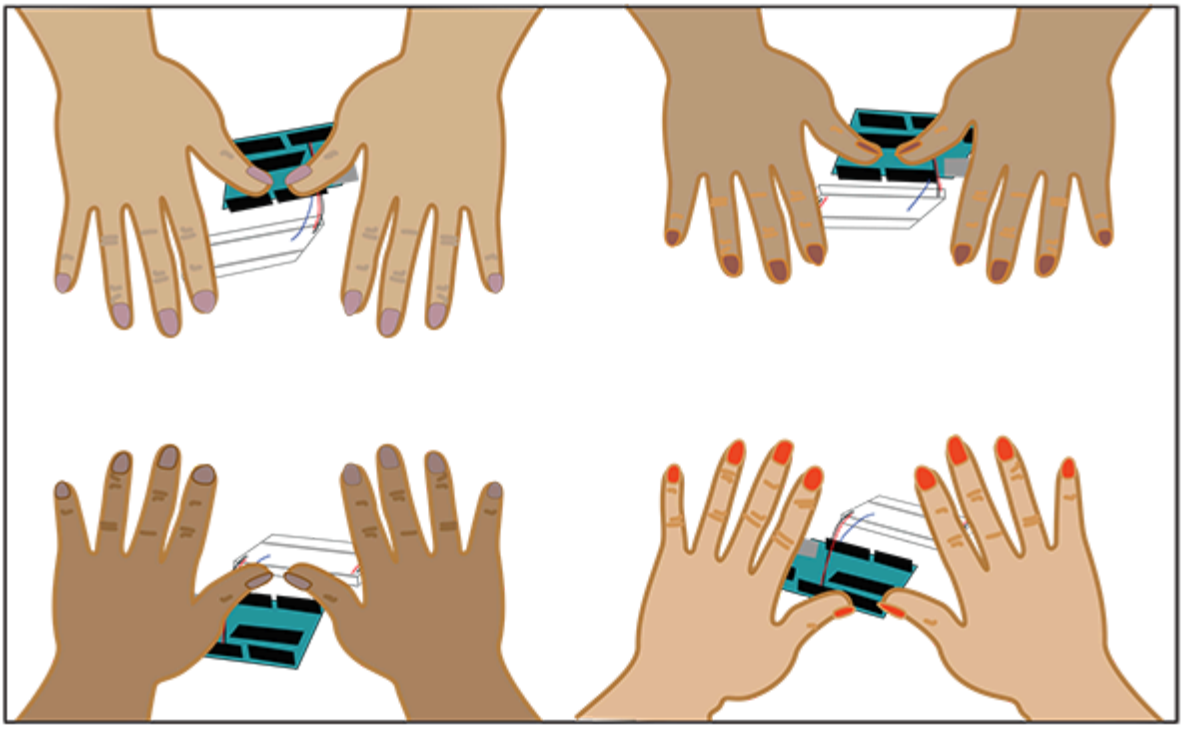


FIGURE 9-3: Get others to try out your projects.

REFLECT AND REPEAT

Now that you have gone through the first pass with your project, you should feel comfortable writing notes for yourself. What went well with your project? What improvements could be made? These notes

can help you iterate on your project and make better versions in the future by improving on past mistakes or false assumptions.

Now that you have a basic understanding and support from some project management techniques, let's talk about common genres for Arduino projects.

A FEW HELPFUL COMPONENTS

We don't have enough room to get into all the varieties of sensors and outputs that exist in the world, but we do want to mention popular choices that can help your projects spring to life.

SENSORS

Here are a few commonly used sensors that can be easily incorporated into your projects.

Sensing Distance and Motion

Passive infrared sensors (PIRs; Figure 9-4) and ultrasonic rangefinders (Figure 9-5) are both used to tell how far away people or objects are from your project. They can also be used to check if someone has walked in front of your project. Since both often give you analog values, you can use these sensors similarly to how you employed the photocell in Chapter 7, "Analog Values."

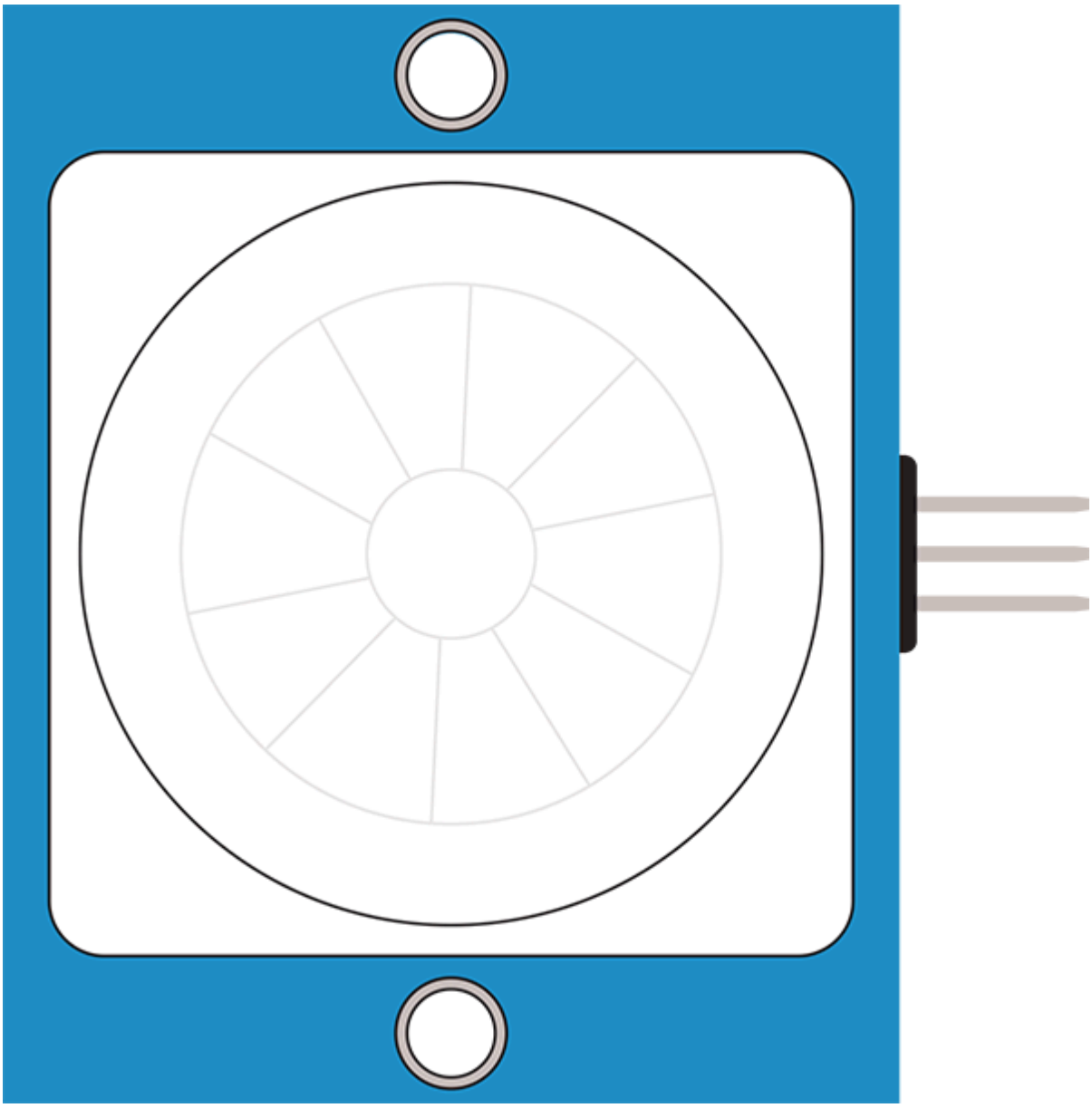


FIGURE 9-4: Passive infrared sensor (PIR)

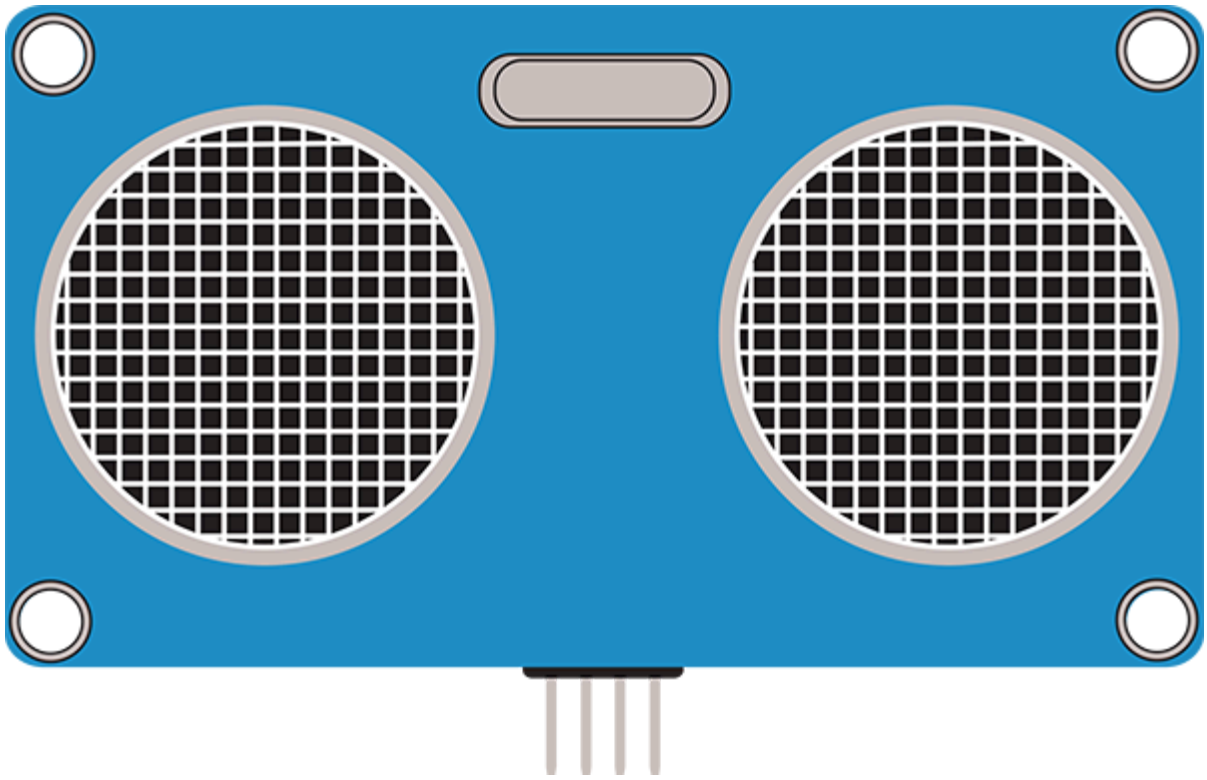


FIGURE 9-5: Ultrasonic sensor

Force-Sensing Resistors

Force-sensing resistors (FSRs) allow you to sense different values of pushing or pressing down on a sensor (Figure 9-6). Since they give analog readings, you can scale the response to move servo motors, light up different sections, or play sounds from a speaker. FSRs are used in gaming controllers and other hands-on interactions. FSRs come in a variety of sensitivities, shapes (including both square and round), and sizes.

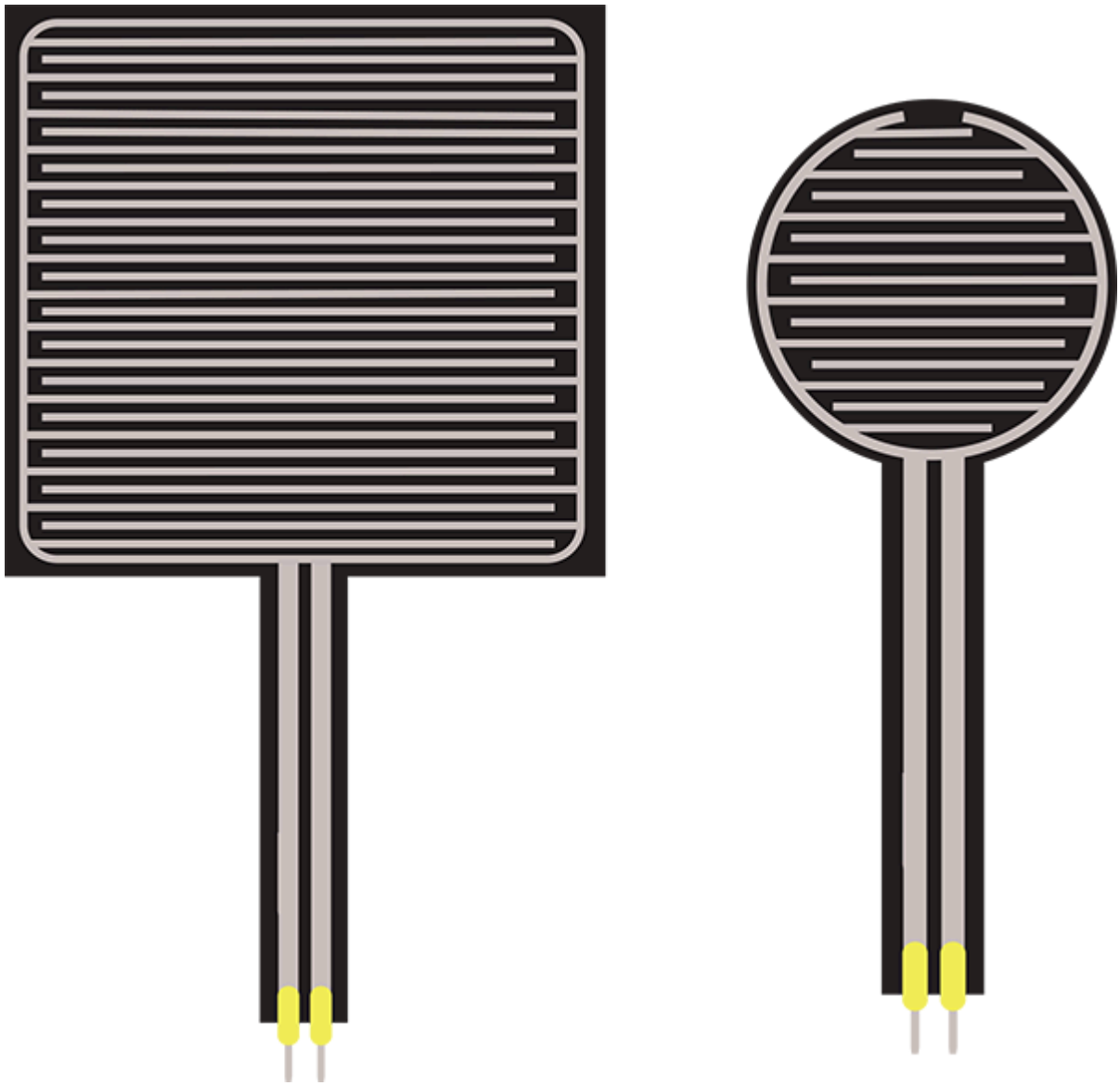


FIGURE 9-6: Force-sensing resistors (FSRs) come in different shapes and sizes.

Other Sensors

As mentioned earlier, there are many more sensors out there that can help extend your Arduino projects. From temperature sensors, to microphones for measuring volume levels, to heart rate and pulse monitors, finding the right sensor can make your project shine.

ACTUATORS AND MOTORS

We have shown you projects that incorporate motion by using servo motors, but there are several other types of actuators (components that can move something) that can make your project move in a variety of ways. We have highlighted a few popular options next.

DC Motors

DC motors come in a variety of sizes and strengths to power even the most stubborn projects (Figure 9-7). They often rotate only in one direction continuously and will move faster or slower, depending on how much power is applied to them (within a safe range). DC motors are used quite successfully to drive wheels, lift heavy objects, and more.

FIGURE 9-7: DC motor

Stepper Motors

Stepper motors (Figure 9-8) are a more controllable type of motor than the basic DC, which means they also require more computing power from the Arduino to function. Rather than turning continuously, stepper motors take single "steps," some percentage of the total rotation. This means that they can be used for accurate positioning and will both start and stop on command. Stepper motors work quite well with the Arduino, though they often require an H-bridge integrated circuit chip or a stepper motor driver to perform more complex behaviors.

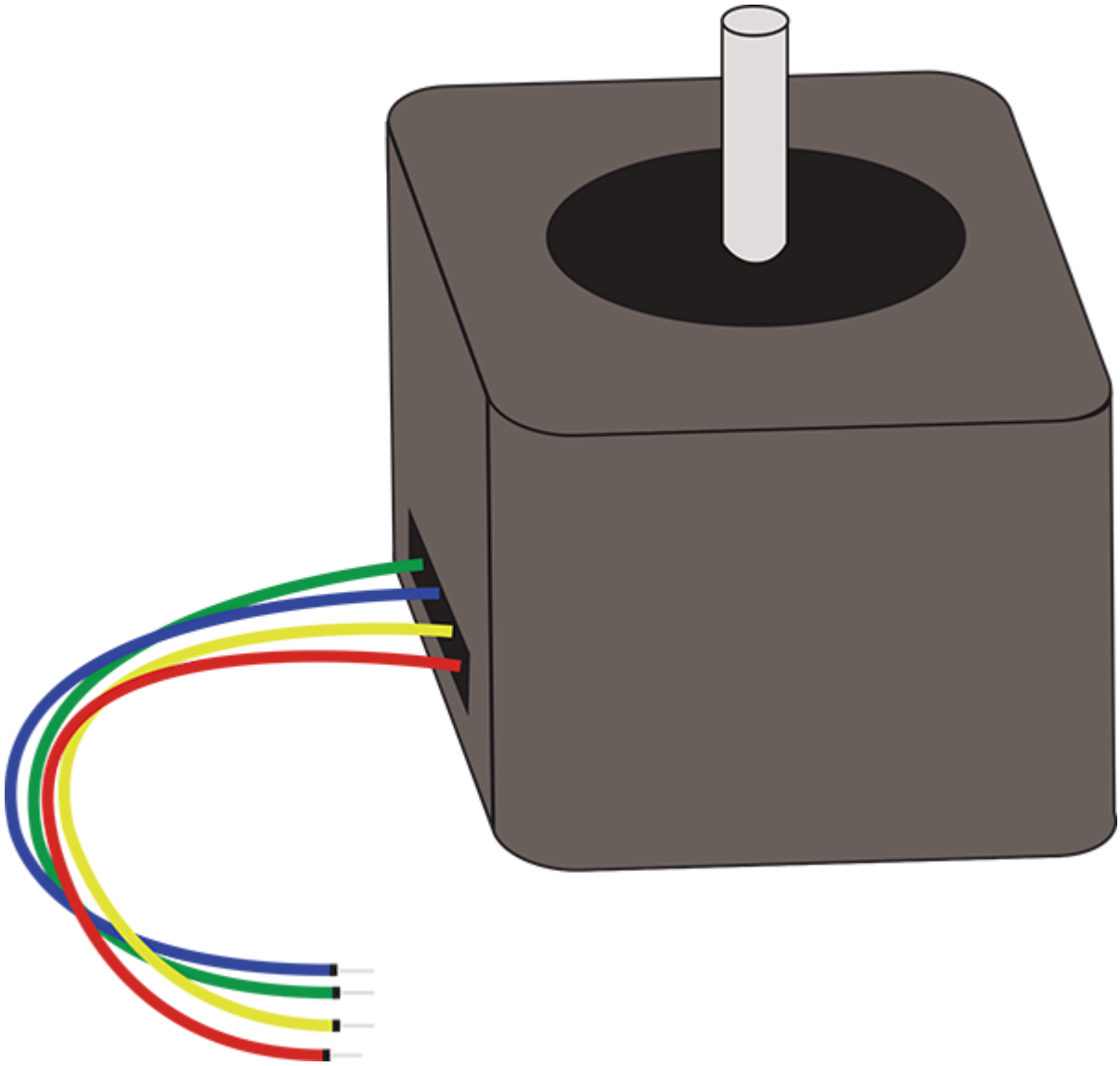


FIGURE 9-8: Stepper motor

Solenoid

Solenoids (Figure 9-9) look quite different from the other actuators we have talked about. Rather than creating a rotation, solenoids are "fired" in a straight line. They have a spring attached to a metal shaft that is either pushed or pulled from the central motor body depending on their type. They are often used in musical instruments to strike percussive or bell-like elements in order to create new sounds.

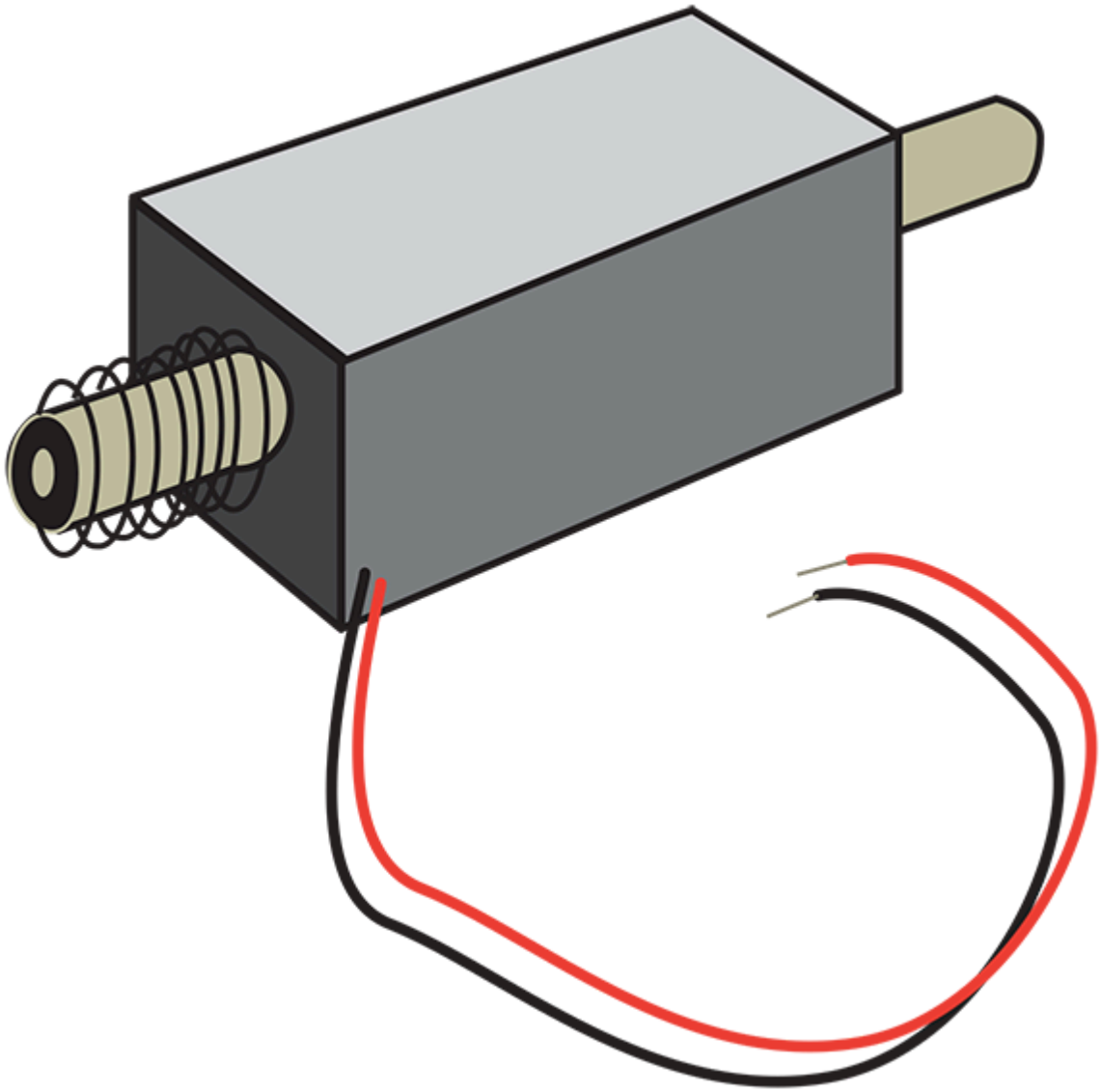


FIGURE 9-9: Solenoid

TYPES OF PROJECTS

We've talked about a wide variety of projects you can build with your Arduino, but we wanted to suggest a couple more genres of projects with a few ideas to help you get started.

HOME AUTOMATION

Although there are a number of products on the market, you can build your own home automation projects using the Arduino. Popular choices for home automation projects include triggering lights, activating fans, or turning off appliances.

ROBOTS

Robots are always a popular choice for Arduino projects. With a few motors and sensors, you can have a pet robot in no time. Single-task robots are also a great choice, from robots that slice butter to those that track objects on the floor. They can even be built out of cardboard (Figure 9-10)!

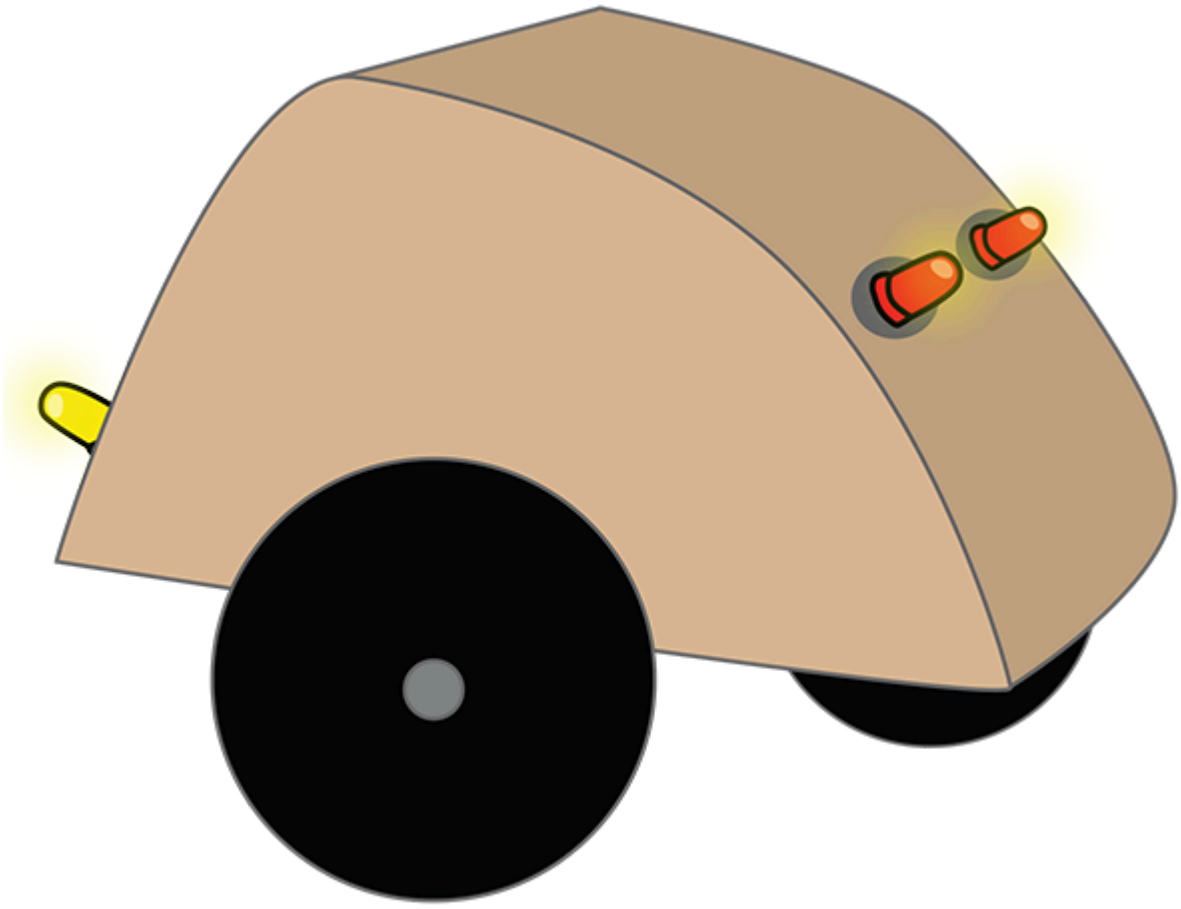


FIGURE 9-10: Cardboard robot pet

WEARABLE PROJECTS

Wearable projects include any sort of clothing, jewelry, or accessories that combine the power of physical computing with portability and accessibility. You can use sensors to get data about your users' pulse or build buttons right into the clothes they wear. Popular projects use gloves, hats, T-shirts, or jewelry and sensors to trigger musical instruments or display screens (Figure 9-11). What type of projects can you think of that use common accessories?

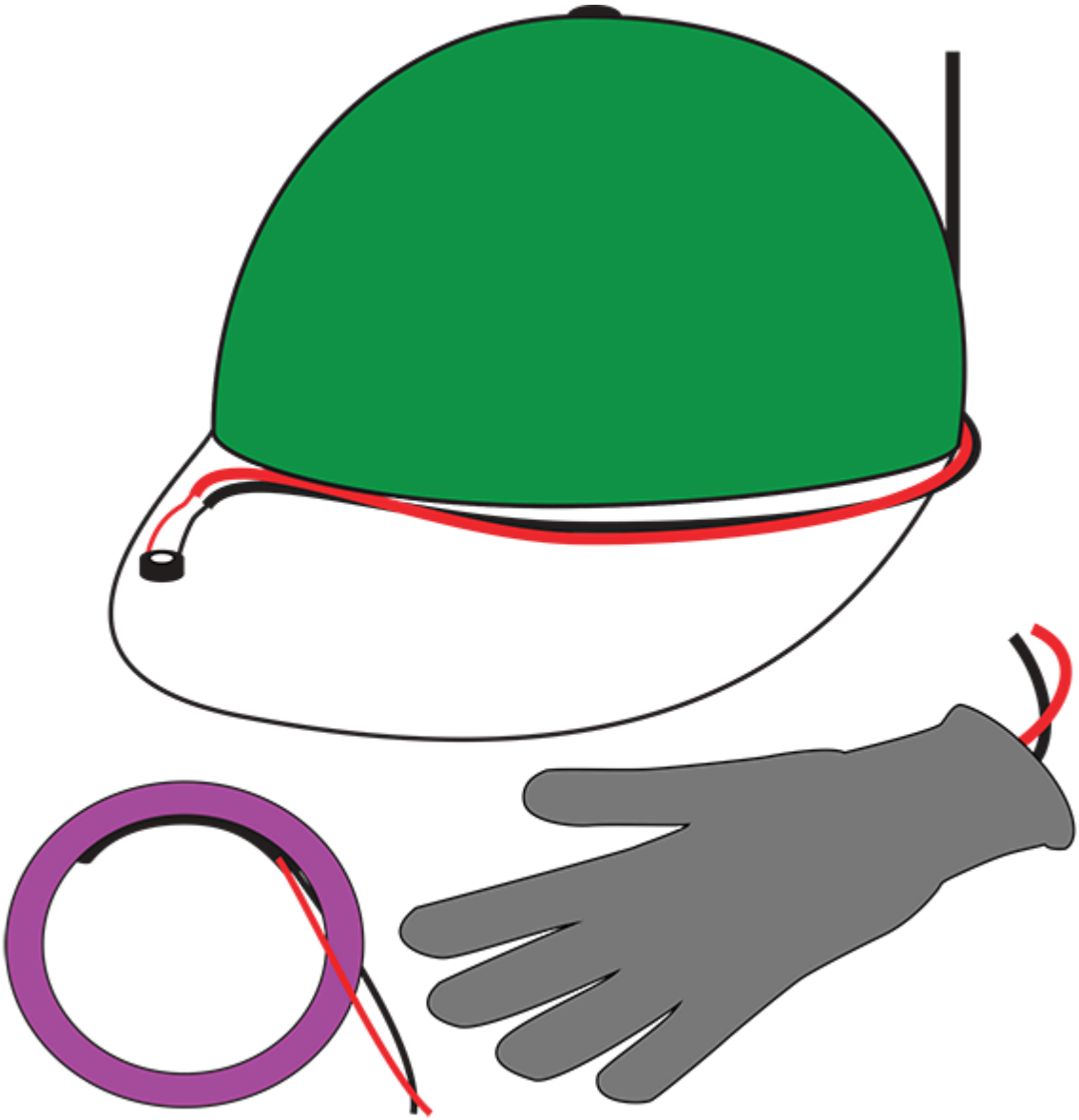


FIGURE 9-11: Bracelets, hats, and clothing items are all popular choices.

ART PROJECTS

Beyond the categories we have already mentioned, you can make any sort of art project you have in mind. From auto-generated painting devices to moving sculpture and interactive books, the only limitation to an art project is your imagination.

OTHER VERSIONS OF THE ARDUINO BOARD

We've mentioned that there are many other versions of the Arduino, which have different functionality. Here is a quick look at a few of the other boards and what they do. There are many more.

THE ARDUINO 101

The Arduino 101 (Figure 9-12) is an excellent choice for moving on from the Uno, since it is the same size and has the same general layout as the Uno. It also has Bluetooth Low Energy (BLE) connectivity and a six-axis accelerometer/gyro. If you want your project to recognize gestures, this might be a good choice. Read more about it here: store.arduino.cc/usa/arduino-101.

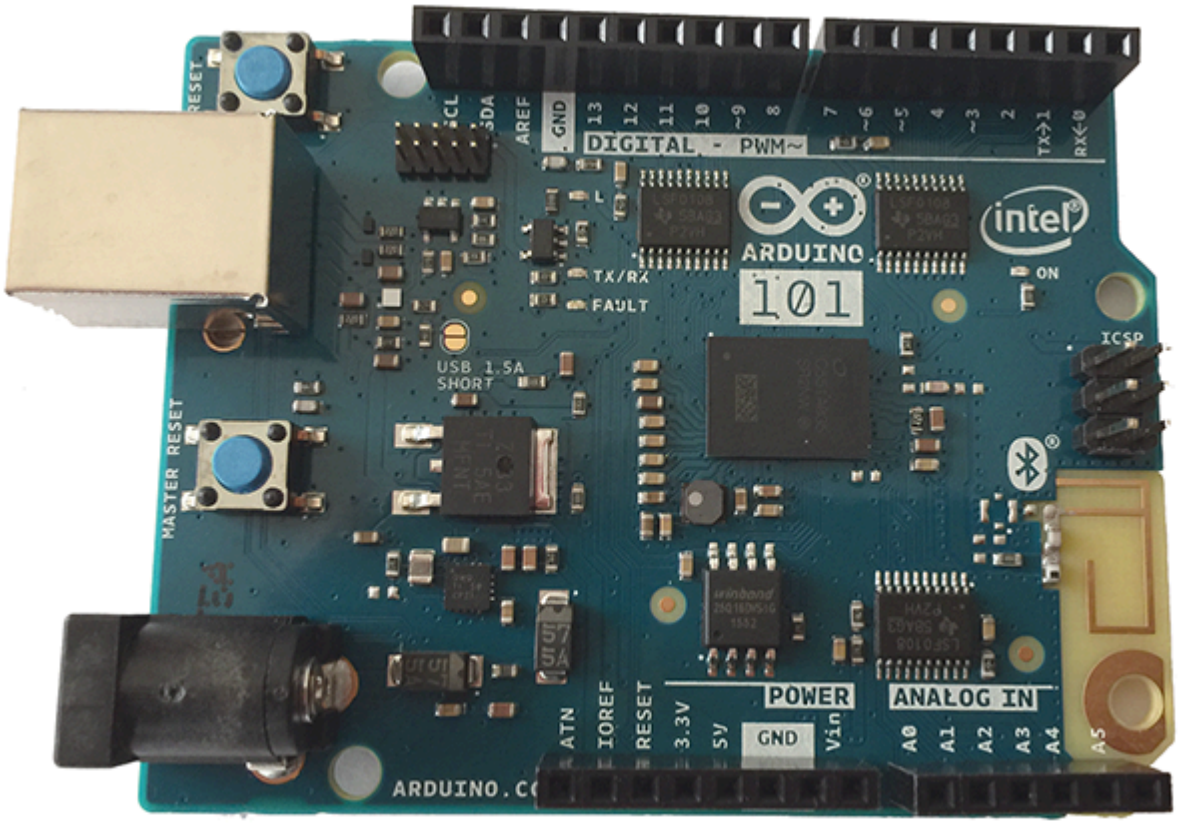


FIGURE 9-12: The Arduino 101

THE ARDUINO YUN

Half Arduino and half Linux computer, the Arduino YÚN (Figure 9-13) will let you use WiFi and the power of an operating system in order to accomplish complicated computing tasks. The YÚN can be used to run Python scripts to analyze data on the Linux side of the board, with the Arduino handling inputs and outputs that respond to that information. It has a slot for an SD card, and both WiFi and Ethernet connectivity built in. More information can be found here: store.arduino.cc/usa/arduino-yun.

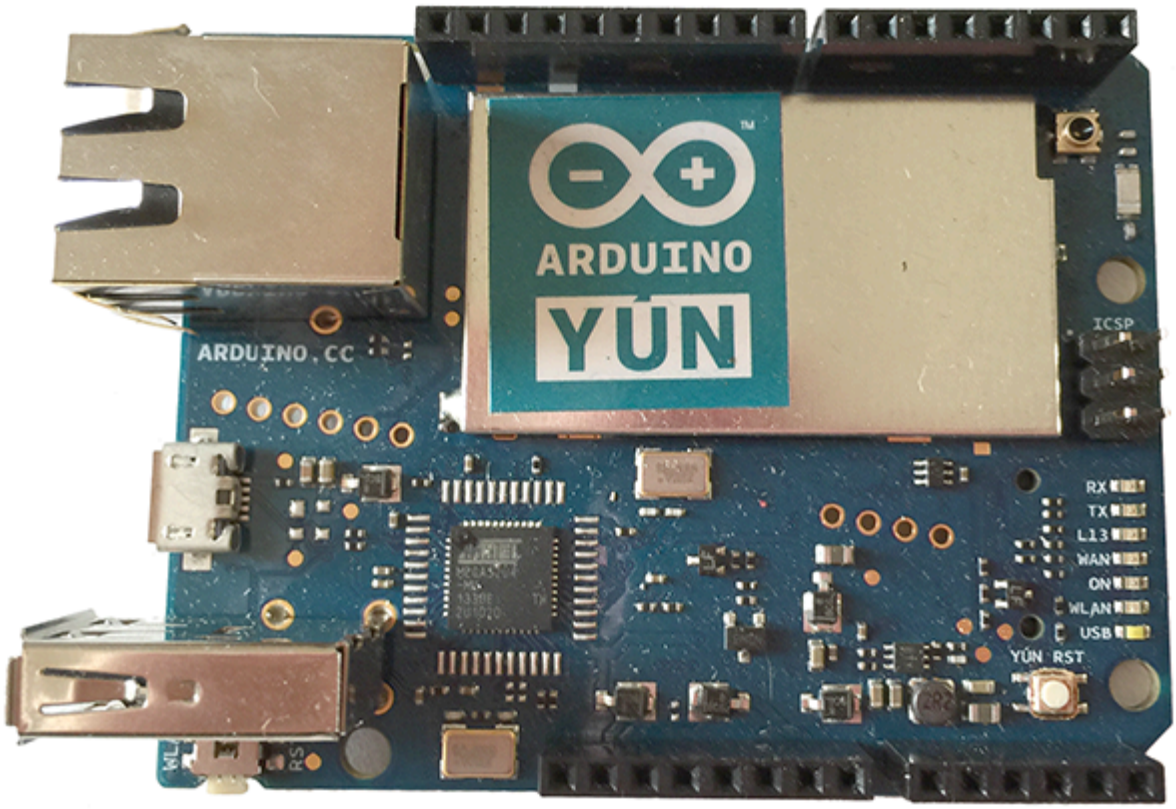


FIGURE 9-13: The Arduino YÚN

THE LILYPAD ARDUINO

As mentioned in the "Wearable Projects" section, sometimes you want the ability to attach an Arduino to a garment meant to be worn, and the Arduino Uno can be a bit clunky. The Lilypad Arduino is great because not only is it flat and less conspicuous, but it can

also use conductive thread in place of wires. This will let you sew your sensors and Arduino directly into the fabric of the project. There are several versions of the Lilypad; Figure 9-14 shows a Lilypad Arduino Main Board.

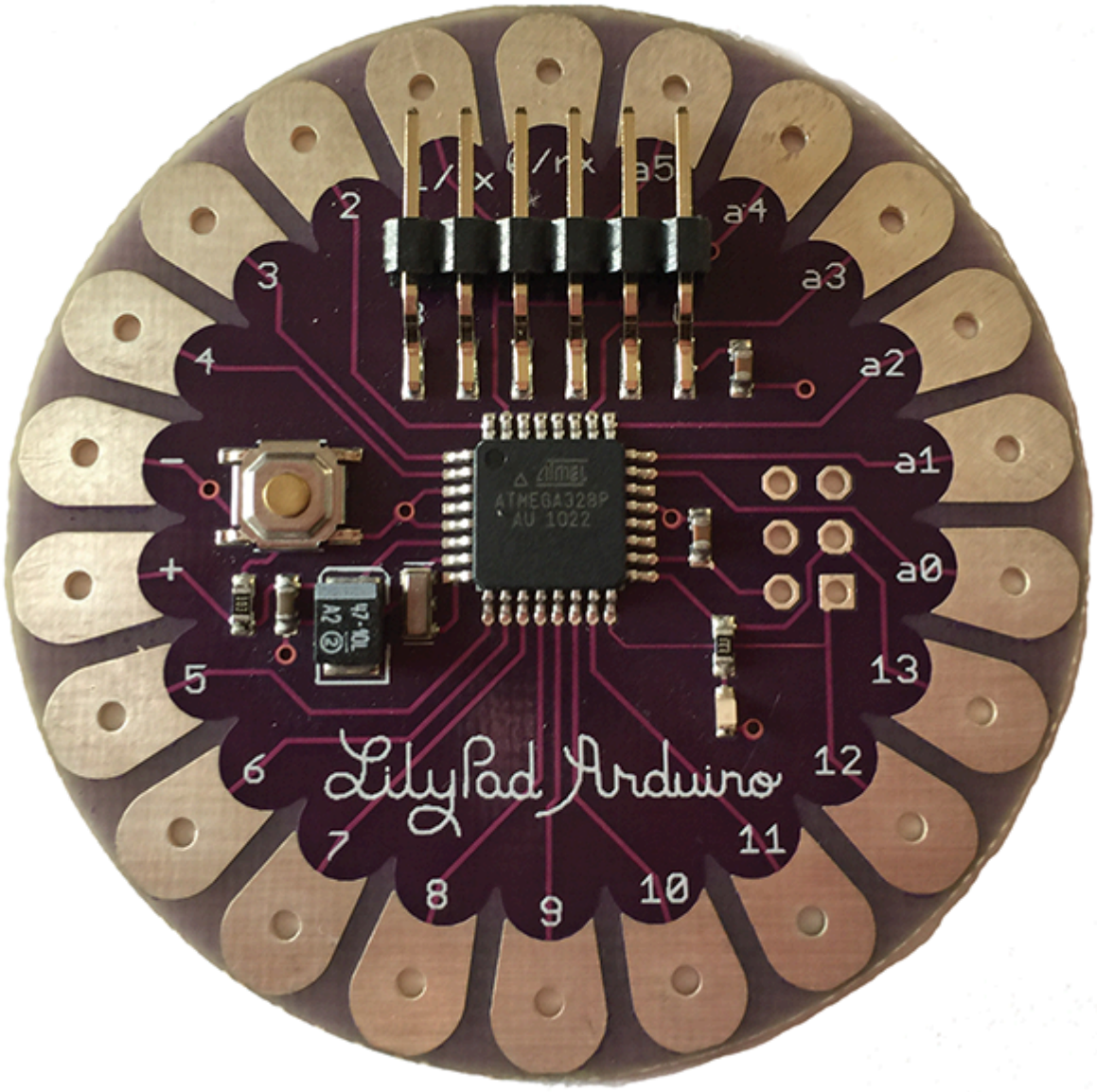


FIGURE 9-14: Lilypad Arduino Main Board

OTHER ARDUINO BOARDS

Although it's beyond the scope of this book to go into every model in detail, we would like to mention a few more boards. The Mega 2560

has 54 digital input/output pins and 16 analog input pins; it is suitable for larger projects. The Leonardo has built-in USB communication, so you can plug in a keyboard and mouse directly. The Micro is the smallest board in the Arduino family, making it appropriate for embedding inside projects. Like the Leonardo, it supports USB communication. The MKR ZERO is a smaller board designed to work with audio applications. The MKR1000 has WiFi connectivity and a built-in rechargeable lithium-polymer battery. The Gemma, developed by Adafruit, is another board designed to be used in wearables.

ARDUINO SHIELDS

In addition to the various Arduino boards, there are a wide variety of branded and third-party "shields" that attach to the top of the Arduino and expand its functionality. These include adding

SD card support in order to save data

Sound file support for playing back recorded audio

Support for controlling motors

and much more.

At arduino.cc/en/Main/Products, you'll find a chart that links to details and technical specifications on each model and on some of the shields that are available.

DOCUMENT YOUR PROJECT AND SHARE IT!

One of the best things about open source projects is seeing what others have come up with, and now it's your turn to share your projects with the world. Here are a few tips that can separate your project from other projects online.

TAKE GOOD PHOTOS

One frustration that often pops up for DIY physical computing projects is that it can be hard to see what is happening in a project photograph. We recommend that you have bright, consistent lighting and a plain background underneath your project.

If you are planning on taking photographs of the wiring, it is extra important that you color-code your wires and avoid crossing them too often. Otherwise, you run the risk of having your project look like spaghetti (Figure 9-15)!

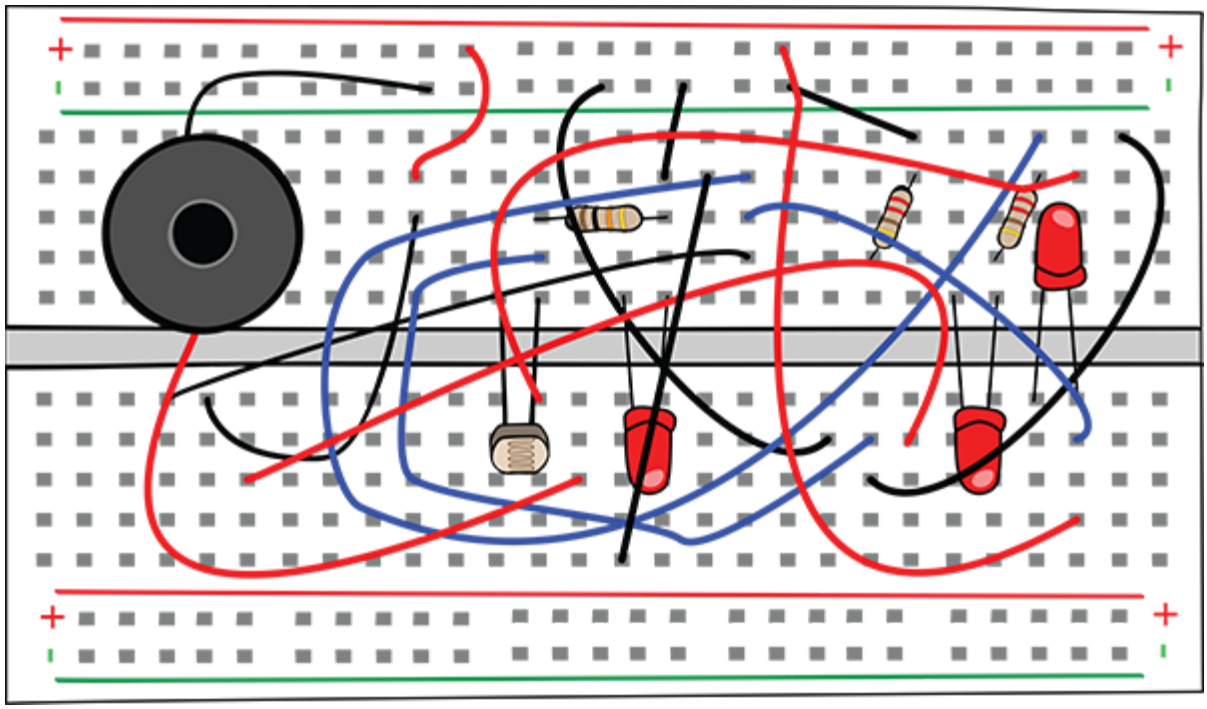


FIGURE 9-15: Spaghetti wiring; this is not an actual circuit!

WRITE UP YOUR PROJECT

If you had problems with some section of code or a certain concept, chances are that the next person who tries to make the same idea (or something similar) will stumble onto the same issues that you did. Writing a summary of your experience building a project or the steps you took to make your project will help you remember the

tricks you have learned for future projects—and may save someone else from a huge headache.

SHARE YOUR PROJECT

Though not required, it can be a great help to share what you come up with for others to see. Many websites, such as makershare.com and instructables.com, have the option to post your own projects and include step-by-step instructions to make them. This is one of the strongest parts about Arduino being open sourced—the knowledge is free to be shared by everyone.

SUMMARY

We've reached the end of this book. In the earlier chapters, you were introduced to basic electronic theory and practice as well as programming concepts. We gave you a few tips on moving forward with your own projects in this chapter. You are now well on your way to building your own fabulous Arduino projects.

A

APPENDIX: READING RESISTOR CODES

If you have just purchased a resistor, it will generally come with some sort of label, but that doesn't help if you find your resistor sitting unaccompanied on a table or in your parts box. Fortunately, every resistor has a set of color bands printed on its casing that tells you the value of the resistor. While there are resistors with six, three, or even one band, the most commonly found resistors by far have four bands, and we are looking at that type in this appendix.

IDENTIFYING RESISTORS BY COLOR BANDS

Let's take a close look at a resistor in Figure A-1. A resistor has two wire leads and a body with color bands on it.

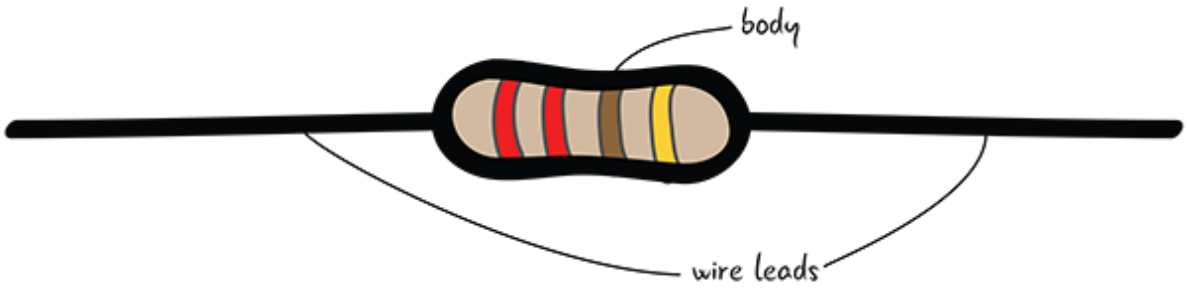


FIGURE A-1: A resistor

ORIENTING THE RESISTOR

Not only do the colors of the bands matter, but also the order in which the colors appear. How do you know what each color means? The first step is to orient your resistor in the correct direction, as shown in Figure A-2. On one side of the resistor, the band color will be either silver or gold. This band should be placed on the right-hand side of the resistor. Look for the silver or gold band on the resistor body and place it on the right-hand side.

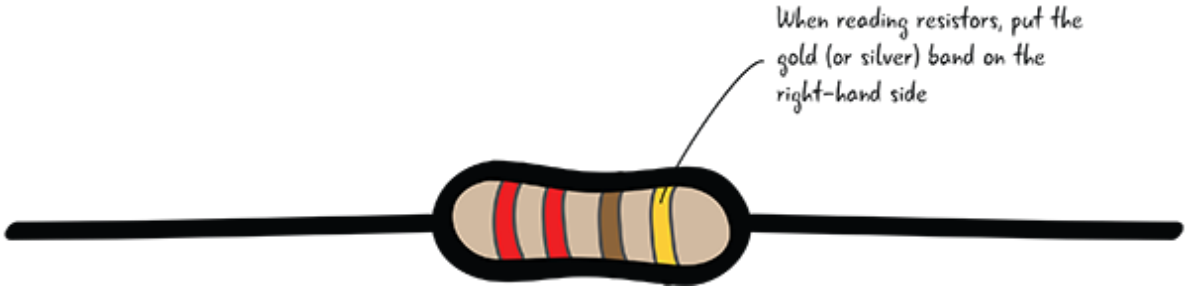


FIGURE A-2: Orient your resistor.

Now that your resistor is oriented correctly, you can identify the other color bands on the resistor body. We have labeled the bands in Figure A-3 in order. The colors on each band have a particular significance.

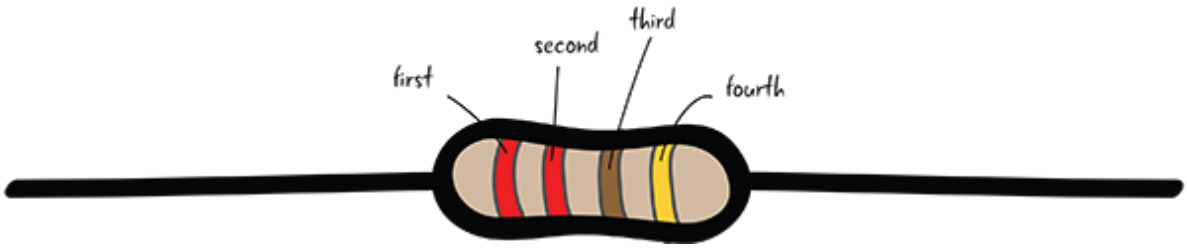


FIGURE A-3: Numbering the bands on the resistor

Resistor Color Chart

Figure A-4 is a standard color chart that all resistors follow. You can find similar charts online. We'll go over what each band means in detail. The colors mean the same thing for all resistors.

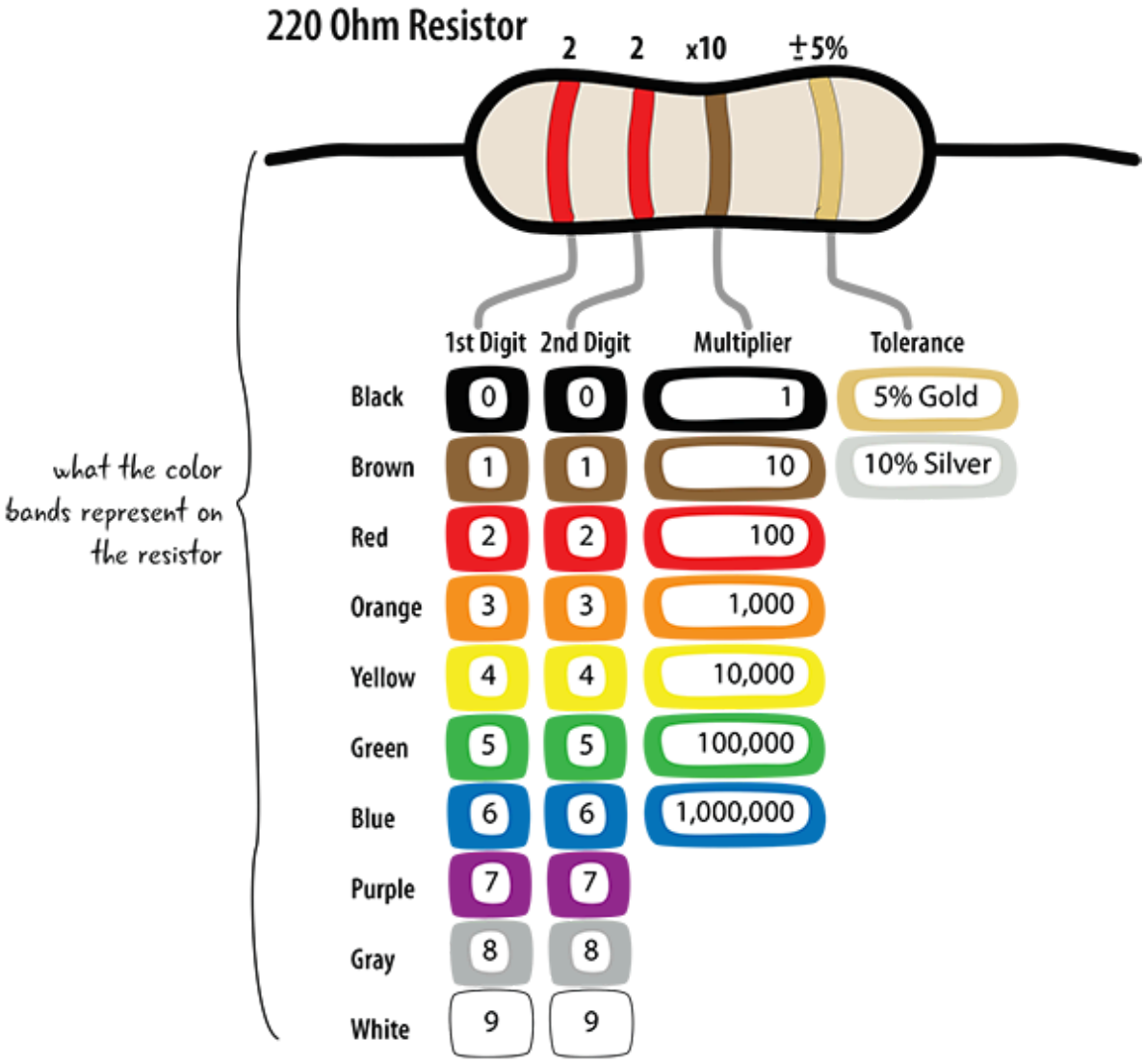


FIGURE A-4: Resistor band color chart

Decoding the Resistor

Now that you've seen the color chart, we'll show you how to apply it to a resistor.

The first band represents the most significant digit, or the first digit in the number. For example, on our resistor in Figure A-5, the first band is red. Looking at the color chart, you see that red on the first band equates to the number 2.

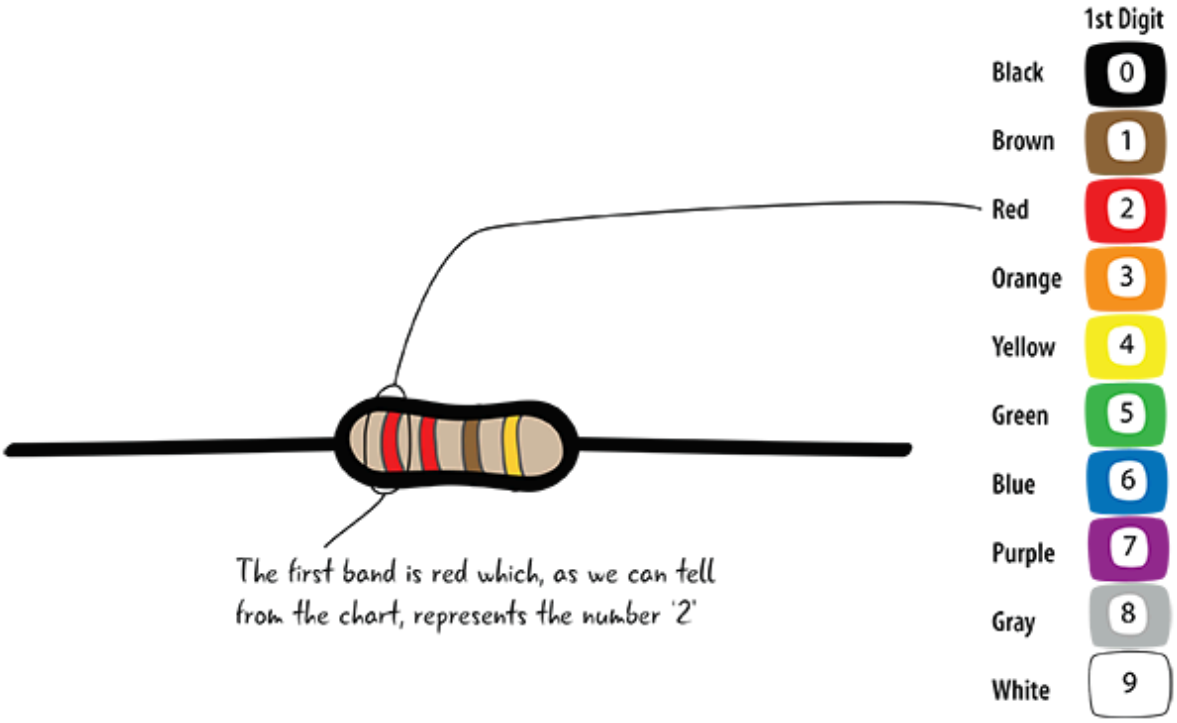


FIGURE A-5: The first band

The second band signifies the second most significant digit. On this resistor, the second band is also red. As you see in Figure A-6, the chart indicates again that the number 2 is represented by the red color of your second band.

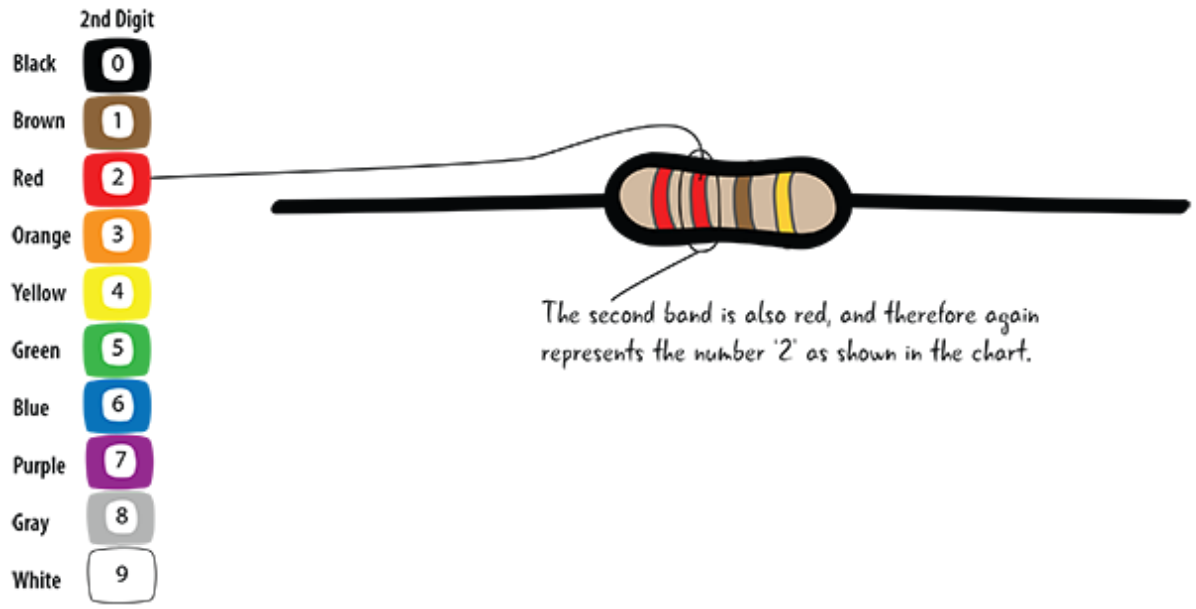


FIGURE A-6: The second band

The first two bands taken together give us the number 22. The first two bands on a resistor will always represent a number between 10 and 99. (We'll explain what these numbers mean shortly.) The third digit is a little bit different.

The third band, shown in Figure A-7, has another meaning. Rather than representing a number like the first two bands, the third band represents a multiplier. This band multiplies the values on the first two bands by a power of 10. We can see this in the third row of the chart in Figure A-4. For this resistor, the band is brown, which the chart tells us means a multiplier of 10, or 10 to the first power. Now that we know these three values, we can calculate the resistor's total resistance using the simple formula shown in Figure A-8: the first two digits times the multiplier equals the resistance (in ohms).

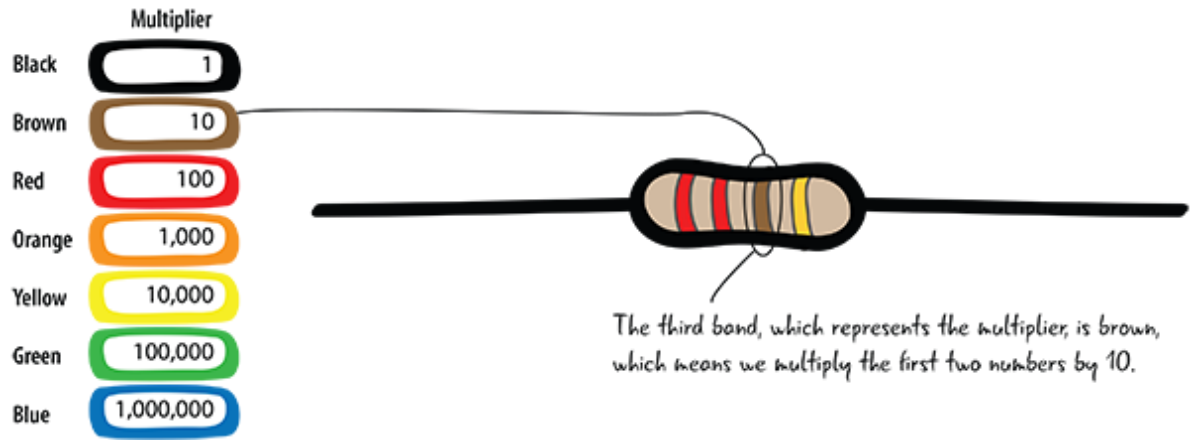


FIGURE A-7: The third band

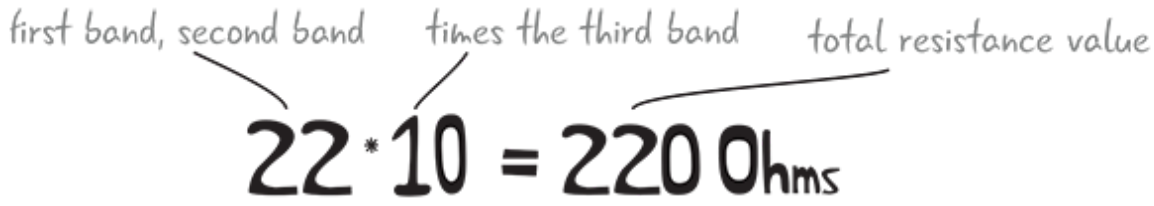


FIGURE A-8: Calculating the value of a resistor

This means that our red-red-brown resistor is a 220-ohm resistor. In fact, all red-red-brown resistors have a value of 220 ohms.

The fourth band of our resistor, Figure A-9, represents the resistor's tolerance or possible range of accuracy. With a gold band, the accuracy is plus or minus 5 percent, which means that our resistor could be as high as 231 ohms (220×1.05) or as low as 209 (220×0.95). (This variation is caused by imperfections in the resistor's manufacturing process.)

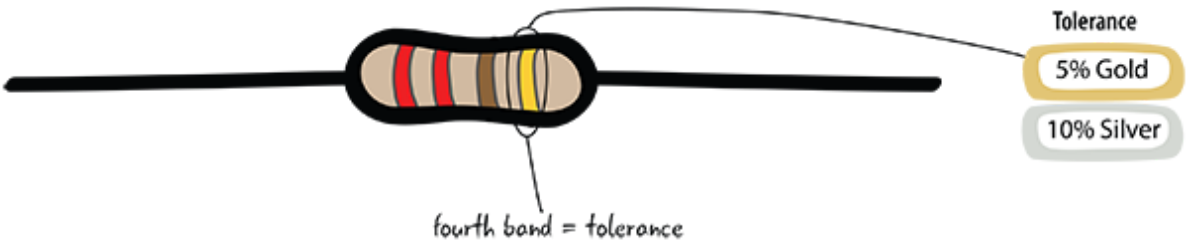


FIGURE A-9: The fourth band

Since the fourth band is always going to be gold or silver, and these are not colors any of the other bands use, we can always use the fourth band to orient our resistor correctly.

QUESTIONS?

Q: Are the band colors universal, and will I have to remember what each color means?

A: All resistors use the same standard color codes we have talked about here regardless of the manufacturer. You don't have to memorize them; you can easily find the color information online, and there are a number of free smartphone apps for all the platforms.

Q: What if the bands are hard to see or they have been painted over or erased?

A: If your resistors are missing the color bands, you can always use a multimeter to confirm the resistance value.

Q: How accurate do I need to be with my resistors?

A: Good question. Hobby electronics and electrical components are not super sensitive to minor variations in resistance. The difference between 209 ohms and 231 ohms is not enough to cause any issue with your LED. However, using a resistor with a much higher rating (double or more) or a much smaller rating (half or less) is enough to cause issues.

Note

Although the four-band resistor is very common, some resistors have a different number of bands. The colors indicate the same numbers in the first three bands, but the tolerance values are calculated differently.

ANALYZING THE COLOR BANDS ON ANOTHER RESISTOR

Let's look at another resistor and evaluate its color bands to figure out its total resistance value. The resistor in Figure A-10 has the color bands brown, black, orange, and gold.

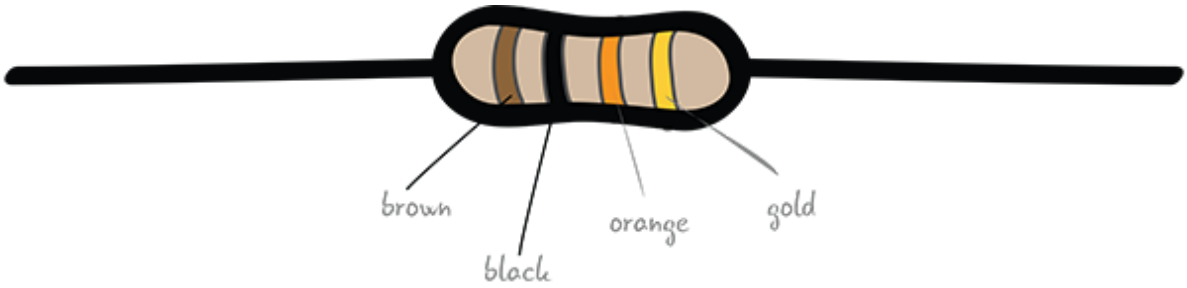


FIGURE A-10: A resistor with the bands labeled

The first step is to orient the resistor correctly. To do that, make sure that the gold band is on the right-hand side (Figure A-11).

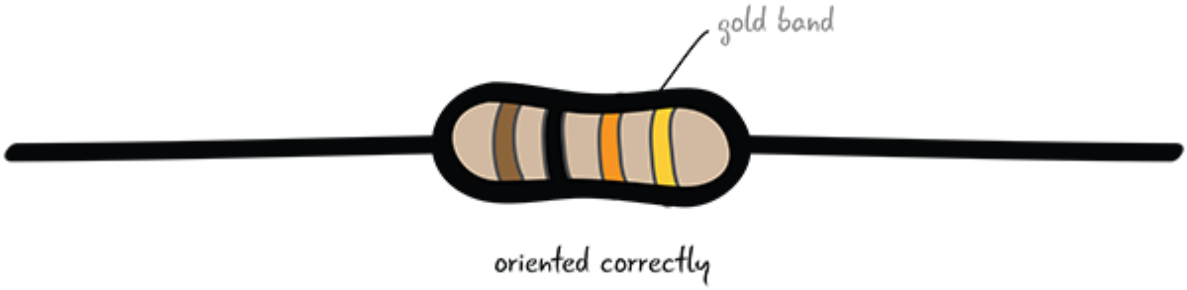


FIGURE A-11: Orienting the resistor

LOOKING AT THE COLOR CHART AGAIN

Refer back to Figure A-4 to reference the color values. You can always look up the chart whenever you need to calculate a resistance value.

Reading the Bands

The first band is brown, so we can look at the color chart and know that the first digit is a 1 (Figure A-12).

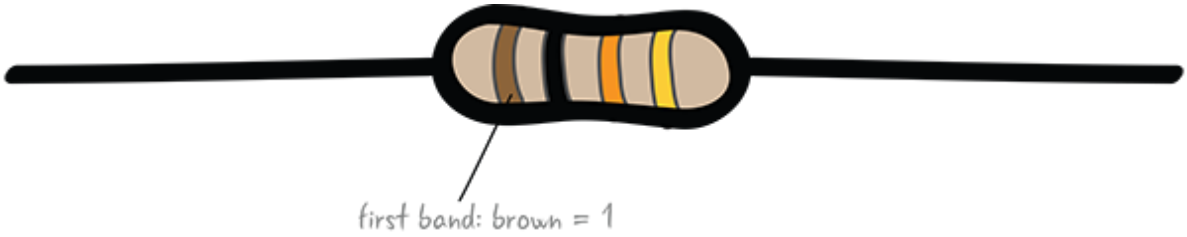


FIGURE A-12: First band

The next color band on the resistor is black, which makes the second digit a 0 (Figure A-13).

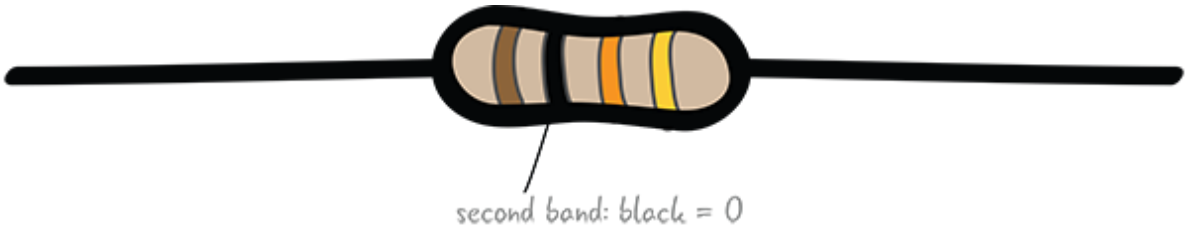


FIGURE A-13: Second band

The third band is orange, which means that the value of the multiplier is 1000 (Figure A-14).

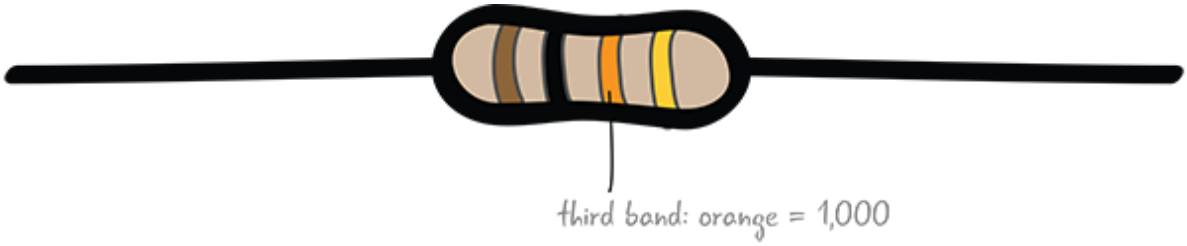


FIGURE A-14: Third band

This means that our resistor value is 10 times 1000. That means our resistor is 10,000 ohms, or more commonly seen as 10 kΩ (Figure A-15).

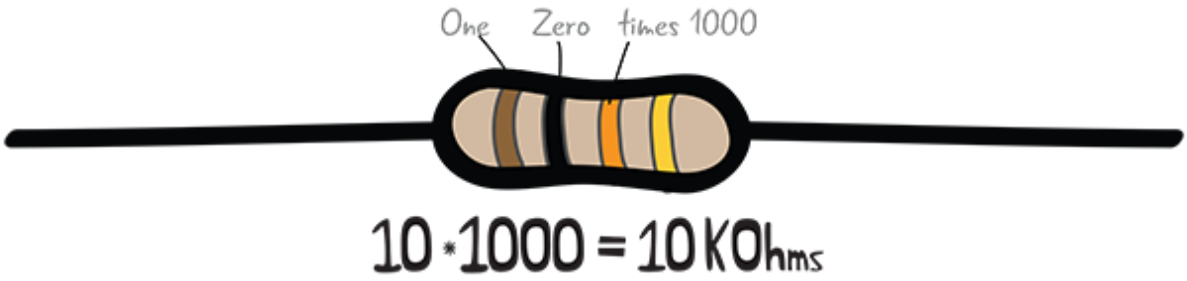


FIGURE A-15: Calculating the resistance in our 10 kΩ resistor using the color bands

INDEX

Numbers

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z