
The Spinwheel
FIELD GUIDE

Welcome!

Welcome to the SpinWheel Field Guide! This guide is designed to start you along the adventure of learning to program your SpinWheel. This book represents just a small fraction of the materials available on our website. We recommend checking out our online content, particularly the interactive tools, in parallel to your explorations with this book! Like a true field guide, you should keep referring back to this book as you move to the online materials and begin to write your own programs for the SpinWheel.

We encourage you to begin by completing our online *Color Coding* adventure at *spinwearables.com/intro*. It is specifically designed to show you how to program the SpinWheel, without having to perform any setup on your computer. Afterwards you can follow the *Initial Setup Guide* to install the software necessary to program the SpinWheel for yourself. Then continue with the various adventures in this Field Guide and online.

Learning a programming language is just like learning any language - it requires picking up some vocabulary, and you might not understand everything at first. With practice, you will become more comfortable with writing your own programs and recognizing what different lines of code do. Don't be afraid to try making modifications to example code or try writing your own code - learning to program is all about experimentation. The more you try, the more you will discover, and the more natural programming will start to feel.

We hope you enjoy exploring physics, math, and art with your SpinWheel through this guide and through our online resources!

Table of Contents

SpinWheel Initial Setup

We're so excited that you are ready to use your SpinWheel! This chapter provides details on the contents of your kit and instructions for turning it on for the first time. You'll learn how to install the Arduino software, which is necessary to program the SpinWheel, and add a new program to your device!

Contents

Your kit contains a SpinWheel, a battery, a micro USB cable and a paper copy of the SpinWheel Field Guide.

The SpinWheel has four main components: a power source (the battery), a sensor (for motion and magnetic fields), a set of lights (the light emitting diodes, or LEDs), and a micro computer (the brain of the device).

The SpinWheel contains sensitive electronic components. While they are securely soldered to the device, they can break if jostled excessively. For this reason, it is important to be gentle with your SpinWheel when attaching the micro USB cable and when putting it into a backpack, purse, or pocket. It is also particularly

important to treat the battery with care; do not puncture or bend it. Storing the SpinWheel in a smal pocket in your bag or in the box you received it in will help keep it safe.

Quick start

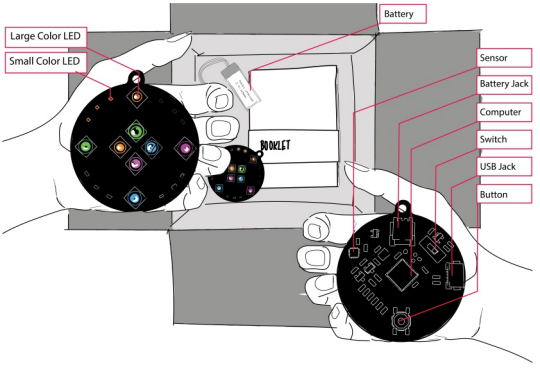
1. The battery of your SpinWheel might not be attached when you receive it. Before attaching the battery, slide the switch on the back of the SpinWheel to be on “USB.” Then firmly insert the battery connector into the battery jack on the circuit board. If you have difficulty, see our troubleshooting guide *(spinwearables.com/troubleshoot)*.

2. To turn on the SpinWheel, flip the switch on the back to "BAT" (for "battery"). You should see it turn on and light up brightly!

3. The SpinWheel comes preloaded with several basic animations. You can press the button on the back of the SpinWheel to toggle between different animations.

4. To turn off the SpinWheel, flip the switch to "USB".

5. To charge, plug a micro USB cable into the USB jack on the back of the SpinWheel, set the switch to "USB" and charge using a computer or USB-to-wall converter. Note: the battery may require charging before use. Reaching full charge takes approximately 1 hr.



Unboxing the SpinWheel

The switch on the SpinWheel should be set to "USB" whenever it is plugged into your computer, whether to charge or to program. You should also keep the switch in this position when you are not using the SpinWheel and for long term storage to protect the battery from discharging.

Installing the Arduino software

Much of the joy of the SpinWheel comes from your ability to change it and make it do whatever you wish! The rest of the chapter will walk you through adding a new animation to your SpinWheel. Do not worry if you find this part challenging. Learning new things can be confusing at first. If you get stuck, check out the troubleshooting guide online at *spinwearables.com/troubleshoot* and don't be afraid to experiment. While feeling confused is normal, it will get easier as you go!

In order to write new animations for the SpinWheel, you will need a way to reprogram its onboard computer. We use the Arduino software to communicate with the SpinWheel.

You can download the Arduino software for free from *arduino.cc/en/Main/Software#download*. For step by step help, Arduino has provided instructions for each operating *system at arduino.cc/en/Guide*.

Configuring the Arduino software

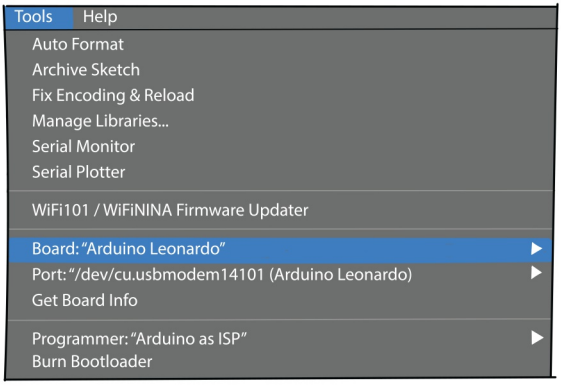
Once the software is installed, we have to configure it to communicate with the SpinWheel.

1. Flip the switch to the position labeled "USB" and then plug your SpinWheel into your computer with the provided micro USB cable.

2. Open the Arduino software.

3. Open the **Tools** menu and go to **Port**. You will see a list of serial ports on your computer; select the port that corresponds to the SpinWheel.

If there are multiple ports and you are unsure which one to use, simply unplug the SpinWheel and see which serial port disappears when you do so. This port corresponds to your SpinWheel's serial port. If you do not see a port appear/disappear, make sure you are using the micro USB cable that came with your SpinWheel as others may not have the needed functionality.



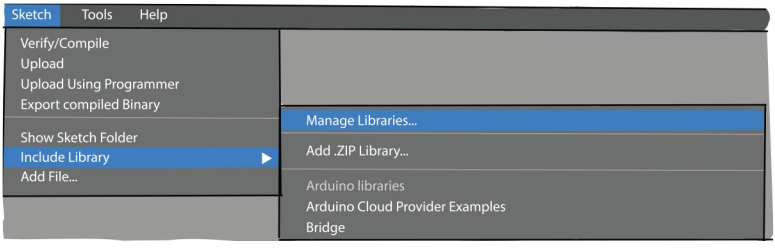
Use **Tools** → **Port** and **Tools** → **Board** to pick the port corresponding to the SpinWheel and the "Leonardo" board type.

4. Go back to the **Tools** menu and select **Tools** → **Board**. Select Arduino Leonardo as the board (a.k.a. processor), so that the software knows which "dialect" to use to talk to the SpinWheel. Computer languages have dialects just like human languages!

Properly selecting the board and port are essential for the Arduino software to communicate with the SpinWheel. If you are unable to upload code to the SpinWheel in the next section, double check that the switch is set to "USB" and that you have the correct board and port selected.

Installing the SpinWheel libraries

1. To get the first set of example programs you can run on the SpinWheel, download our SpinWearables Arduino Library using **Sketch → Include Library → Manage Libraries....**



Installing Arduino libraries

2. In the search bar of the Library Manager, search for **SpinWearables** and then click **Install**.

3. You will be automatically prompted to install two other required libraries (**NeoPixel** for controlling the LEDs and **ICM 20948** for reading the motion sensor). You will need to install both of these to use the SpinWheel.

Running a program on the SpinWheel

To test that your SpinWheel is working properly, you can install a new program, or sketch, from the example files to animate your SpinWheel.

1. Choose a file to install by opening **File → Examples → SpinWearables** and picking one of the examples. For instance, pick `BlinkingFirmware`. This will open a new window with the code.

2. Upload the code to your SpinWheel by pressing the upload button (the arrow at the top).

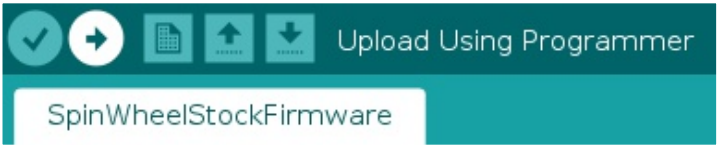
Now your SpinWheel will have the new colorful blinking pattern (from `BlinkingFirmware`) you just uploaded.

Sometime, while uploading code, you might get an error. While error messages will contain some information about the cause of the error, figuring out how to fix it can seem really overwhelming. Always begin by carefully reading the error, attempting to determine the problem. The troubleshooting guide online can be used to find solutions for some common problems. This might seem overwhelming at first, but with some practice you will learn to disregard the unimportant noise in the error message and focus on the one or two words that point out the actual problem.

Feel free to open any of the other SpinWheel sketches and upload them onto the device. Do not worry about understanding what the code does, you will learn more about this language in future lessons. We encourage you to experiment with these examples! If you want to save any changes, you will be prompted to save the sketch in a new location (can be anywhere on your computer). The original file will always be available to open again.

Uploading a new sketch to your SpinWheel will overwrite the sketch that came on it. If you want your SpinWheel to have the original sketch again, simply open the `SpinWheelStockFirmware` example and upload it.

In future SpinWheel activities, you will be writing new sketches to animate the SpinWheel. To transfer a sketch from your computer to your SpinWheel, simply connect your SpinWheel to your computer, change the switch to "USB", open the code of your new sketch in the Arduino software and press the upload button.

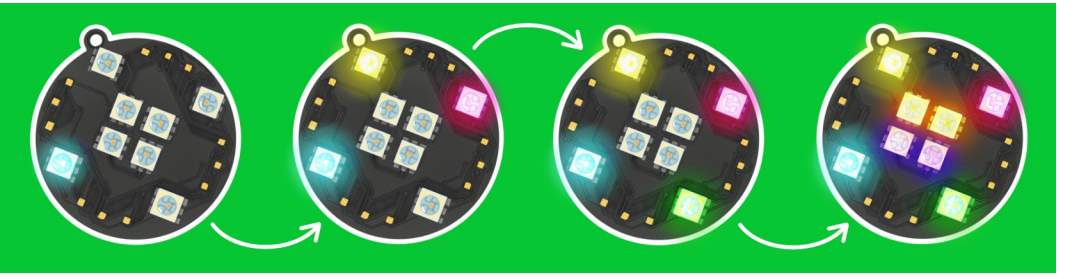


Upload programs to your SpinWheel using the **Upload** button (highlighted in white).

Congratulations! You are now ready to continue with the rest of the SpinWheel activities!

This SpinWheel Field Guide contains some hands-on adventures and reference material. This material is found in expanded versions online, along with many more activities for you to enjoy. We highly recommend that you make use of both resources. In addition, the online version includes virtual SpinWheels that allow you to test and experiment with your code and see what the result might look like before uploading to your real SpinWheel!

In particular, you might be interested in the *Biology of Sight* group of webpages if you would like to learn more about color and how to program the SpinWheel's display. Then the *Intro to Animation* and the related activities can teach you how to use a computer program to animate a display. Finally, the *Dancing with Color* set introduces the use of the motion sensor, together with some introductory lessons on the physics of motion. Beyond this central group of ideas, the online resources also cover many additional skills and hacks around the SpinWheel.



Customizing the SpinWheel's Display

Now that you have successfully added a sketch to your SpinWheel, let's learn how to create a simple program. In this chapter, you'll write a program (a set of written commands for your computer to follow) that lights up your SpinWheel's large LEDs.

Approach this page the way you would approach the first few lines of a foreign language you want to learn. Try to pick out words that make sense, without worrying about the overall structure or proper grammar. As time passes and you have learned new things, come back to this page and see whether you can understand a bit more of it.

Computers follow instructions. They do not solve problems on their own. So, when writing a program for the SpinWheel's onboard computer, you need to be very explicit in the instructions that you write! The particular language we are using requires our programs to have a certain structure. In the Arduino software, if you navigate to **File -> New**, you can see the basic structure for an Arduino program. It includes two basic sections: setup and loop.

To produce a program capable of sending instructions to the hardware of the SpinWheel (e.g. the LEDs and motion sensor), we need to add a few more lines to the new sketch (If you are curious about why you need these

other lines, then check out *Arduino 101*.):

```
#include "SpinWearables.h"
using namespace SpinWearables;

void setup() {
  SpinWheel.begin();
}

void loop() {

}
```

Now we can add something to the `loop()` section of our sketch to make the SpinWheel light up. You may have tried this with the virtual SpinWheel in our online *Color Coding* adventure. To start with, let's turn all the large LEDs purple by simply adding two commands to the empty line in the loop section of code we started above and upload it to the SpinWheel.

```
SpinWheel.setLargeLED(0, 255, 0, 0);
SpinWheel.drawFrame();
```

If you don't want to have to type this into the Arduino software, you can also open the code from **Examples -> SpinWearables -> Paper_Guide -> Purple_LEDs** in the Arduino software and upload it to your SpinWheel.

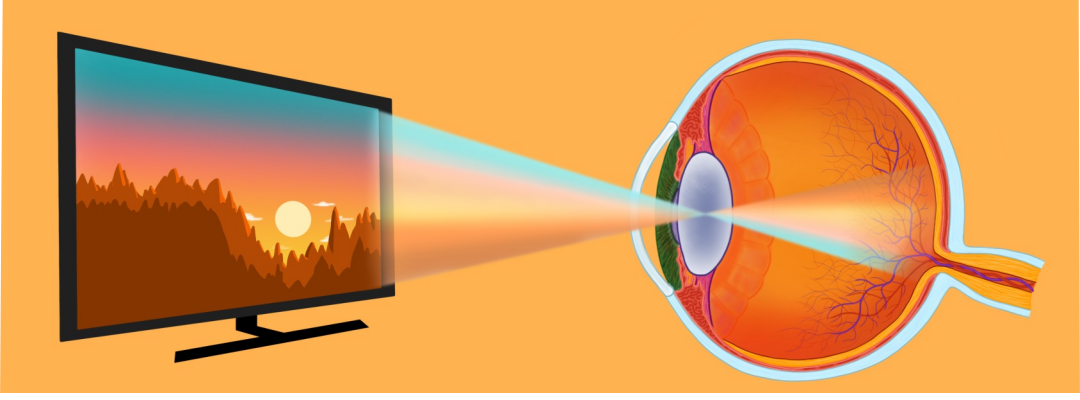
Sometimes when we write a sketch, it won't upload. It might be because we forgot a semicolon or another typo. Learning to find these errors can feel overwhelming at first. If this sketch fails to upload, first check that you have the correct `port` and `board` selected and that you have copied the header information from above. For more suggestions, check out the troubleshooting guide on our website.

You may remember from the *Color Coding* webpage that the command we wrote will change the color of the large LEDs based on the values given in the command. These values follow the format of `SpinWheel.setLargeLEDsUniform(amount_of_red, amount_of_green, amount_of_blue)`. In this case, it will light up all of the large LEDs purple. In the next chapter, we'll

explore more about how we create a rainbow of colors using just red, green, and blue light.

If you want to do something more complicated, then try replacing `SpinWheel.setLargeLEDsUniform()` with your code from the *Color Coding* page (*spinwearables.com/intro*). These commands are also listed in the back of this guide in the *SpinWheel Functions Reference*. Keep experimenting like you did on the virtual SpinWheel to create beautiful designs from your imagination on the SpinWheel's interface!

SpinWheel LED numbering

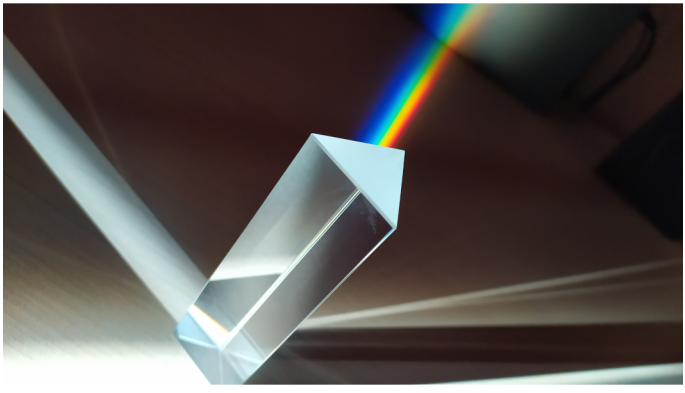


Mixing Color with Light

Human perception of light and color has many curious features rooted in biology and physics. In this chapter, you will learn how to trick your eyes into perceiving a rainbow of colors using only red, green, and blue LEDs.

When light comes from the Sun (or most other sources of illumination), we perceive it as lacking a hue (we call this white light). In reality, white light is made up of many colors. You can use a prism to separate the components of the mixture. A prism works by bending, or "refracting", light at different angles depending on its color, thereby allowing us to see all of the colors that make up white light. This is why if you let sunlight shine through a prism, you can see a rainbow.

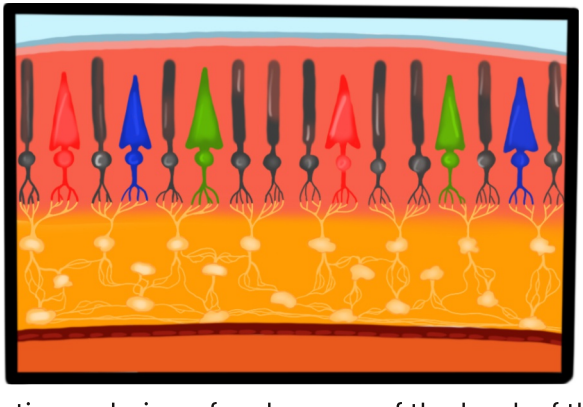
The light-sensing tissue in the back of our eyes, the retina, has small cells that respond to some of these colors. They are called "cone cells" and are classified into three separate groups by the color that they sense the best: red, green, or blue. Each of these groups of cells responds to one of these three colors, but not the others. For instance, the blue-sensing cones respond to blue light, but they do not respond to red light, and vice versa.



White light being split into colors by a prism. The white light shines on the prism from the bottom left, and a big part of it is refracted and split as it passes through the prism.

If our eyes can sense only red, green, and blue, how can we see yellow? Our eyes and brains have evolved so that our red- and green-sensing cones both respond slightly to yellow. If our brain detects that both groups of cones are activated, it knows to interpret the color as yellow. We can see other colors this way too. For instance, purple activates both red- and blue-sensing cones.

We can exploit this imperfection of our eyes to make rich colorful electronic displays while using only three colors. For instance, since our eyes cannot distinguish between true purple and a mixture of blue and red, we



An artistic rendering of a close-up of the back of the eye showing the rods (black) and cones (triangles colored by type).

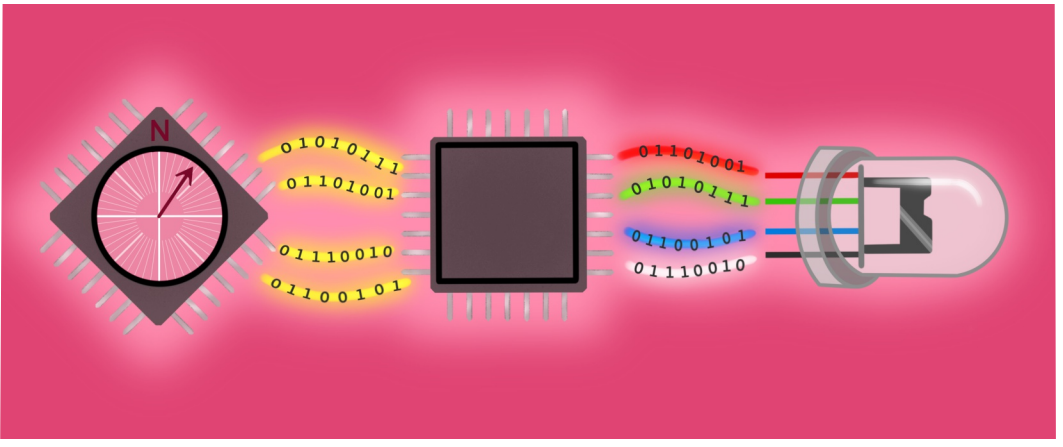
don’t need a purple light source, only blue and red lights (and green for the other color combinations).

The SpinWheel's colorful display takes advantage of this. If you look closely at an LED light on the SpinWheel, you can see that it contains 3 small light sources placed very close together: one red, one green, and one blue. Combining these lights in different intensities allows for a wide variety of colors to be displayed on the LEDs.

To better see the components of an LED on the Spinwheel, open **Examples -> SpinWearables -> Booklet -> Four_LEDs** in the Arduino software, and upload it to your SpinWheel. Look closely at the red LED on the SpinWheel; you should see 1 red light in the LED. Likewise, the blue and green LEDs will also have 1 light in them. If you look instead at the white LED, you should see 1 red, 1 green and 1 blue light inside it. When each light is turned on, the colors combine to make white light.

At this point, we encourage you to try out the *Biology of Sight* adventure online (*spinwearables.com/sight*). You’ll learn more about how your eye works and experiment with coloring the SpinWheel’s LEDs. If you are wondering why mixing color with paint (instead of light) is different, then we recommend you check out our online lesson on *Color Theory* (*spinwearables.com/colortheory*).

An up-close picture of an LED, showing red, green, and blue subpixels.



Arduino 101

In this chapter, we'll dive into each line of the code you used to color your SpinWheel's LEDs in the earlier chapters. We'll expand on how to write an Arduino sketch and explain how to monitor the output from the SpinWheel's motion sensors. We hope you will keep coming back to this chapter as a reference while you work through this book and start the online adventures.

Many simple computer chips for DIY projects use the Arduino software, a platform for writing and uploading code onto a physical device, like the SpinWheel. The Arduino software uses a simplified version of C++. We will begin to introduce this programming language below.

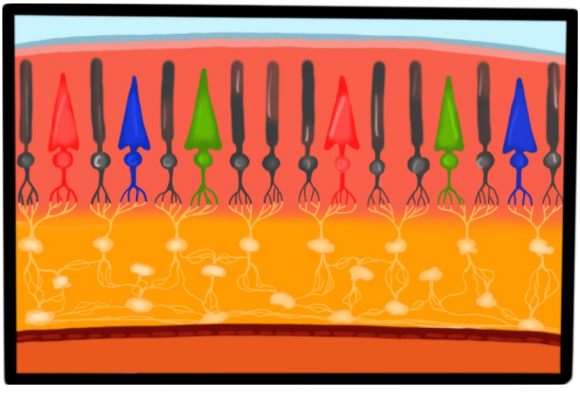
The skeleton of an Arduino program

As we discussed in Customizing the SpinWheel's Display, the Arduino software requires our programs to have a certain structure. The most basic program looks like this:

```
void setup() {

}

void loop() {

}
```

This program does absolutely nothing. It contains two blocks (or sections) of code which start and end with the brackets: `{` and `}`. For our program to do anything, we need to fill these blocks with instructions. The first block is called `setup` and runs only once, immediately after the device is powered up. This block is used for setting up any initial conditions the SpinWheel might require.

Next, there is the `loop` block. This block is executed repeatedly "in a loop", starting immediately after `setup` is done. The loop repeats itself until power is turned off. Most of our instructions will be written in this block. They will frequently involve measuring time or motion with the SpinWheel and then producing a colorful pattern based on these measurements.



When you turn on the SpinWheel, `setup()` is run once and then the `loop()` block is run repeatedly until the SpinWheel is turned off.

Extra elements for a SpinWheel program

To produce a program capable of sending instructions to the hardware of the SpinWheel (e.g. the LEDs and motion sensor), our program requires a few more lines:

```
// Include extra resources and commands
// specific to the SpinWheel.
#include "SpinWearables.h"
using namespace SpinWearables;

void setup() {
  // Instruct the specific SpinWheel hardware
  // to be ready to receive instructions.
  SpinWheel.begin();
}

void loop() {

}
```

This piece of code starts with a comment. Any line that starts with two slashes (`//`) indicates an explanation for the person reading the code. As you look at other sketches, they will help you figure out what the commands for the computer do.

Adding `#include "SpinWearables.h"` and `using namespace SpinWearables;` before `setup()` ensures that the rest of the program has access to the extra resources and commands specific for programming your SpinWheel.

It is also necessary to add `SpinWheel.begin();` in the `setup` block. This line makes sure that the SpinWheel hardware is ready for the instructions that we will add in the `loop` block of code. When you start writing other programs for the SpinWheel, these extra lines will be essential.

The `loop` block is still empty, so this program still will not do anything interesting. However, our `setup` section is complete; it prepares the SpinWheel to receive instructions. Through our activities, we will rarely need anything more sophisticated in `setup`.

Finally, let us turn on an LED by adding a single function (or command) in the loop section of code we started above.

```
SpinWheel.setLargeLED(0, 255, 0, 0);
SpinWheel.drawFrame();
```

Open the code from **Examples → ...?????? → Red_LED** in the Arduino software and upload it to your SpinWheel. It should cause one large LED to turn on in bright red.

Let's discuss each line we added to the code:

The first line, `SpinWheel.setLargeLED(0, 255, 0, 0);` tells the SpinWheel to get ready to set one LED to the color specified. The first item (also called a "parameter" or "argument") in the parentheses identifies the affected LED and should be between 0 and 7 (the programming language used by Arduino starts counting at 0). The other three numbers are the red, green, and blue components of the desired color. They have to be numbers between 0 (color is off) and 255 (color is on at full brightness). Together, this line of code looks something like: `SpinWheel.setLargeLED(LED_you_want_to_change, amount_of_red, amount_of_green, amount_of_blue)`.

The line `SpinWheel.drawFrame();` signals to the SpinWheel that we are done specifying actions to take. It tells the SpinWheel to "draw" all of the commands that were listed above it. Without this line, the LED specified in `SpinWheel.setLargeLED` won't light up.

Additional programming notes

While looking at the code in the Arduino software, you may have noticed a few more things about the style of this programming language:

– We tend to have only one "command" per line. This makes the code more readable.

– Each command is followed by a semicolon ; . That makes it easier for the computer to separate different

commands.

– Commands take the form of their name (e.g. `SpinWheel.setLargeLED`) followed by parentheses `()`.

– Inside these parentheses, we frequently put some extra information: this information can control how a command performs. For instance, in `setLargeLED` we have one parameter that selects the LED we want to modify and three parameters for the color of that LED.

– There are other ways in which LED colors can be modified and motion be detected. We will be discussing many such tools in later chapters and in the online material.

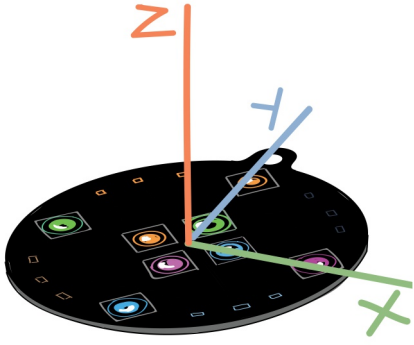
Receiving communication from your SpinWheel

It can be very useful to have a way to receive messages from the computer chip (in our case, the chip on the SpinWheel) you are programming. For instance, you might want to see the values that the motion sensors are recording. Having this information can also be very important when debugging attempting to find errors in some code (commonly called debugging) .

Different computer languages provide different ways of receiving messages from a computer chip. If you are working with Arduino, then you can use the **Serial Monitor** to see the messages being sent from your SpinWheel to your computer over the micro USB cable. To access the **Serial Monitor**, navigate to **Tools** → **Serial Monitor** in the Arduino software.

For example, using the Serial Monitor, we can check the output of the SpinWheel's motion sensors. The sensors are capable of detecting a magnetic field, acceleration, and rotation. To start with, we will look at rotation. In the *Step Counter* and *Dancing with Color* online adventures, we use

this output in more exciting ways. Keep reading to learn more and refer back here when you start these adventures.



This picture demonstrates the three axes that the SpinWheel can detect rotation around.

The code below uses the Serial Monitor to record the rotation of the SpinWheel in one dimension, corresponding to the z-axes in the above diagram. To see this rotation, try holding the SpinWheel flat on the table and spinning it on the surface.

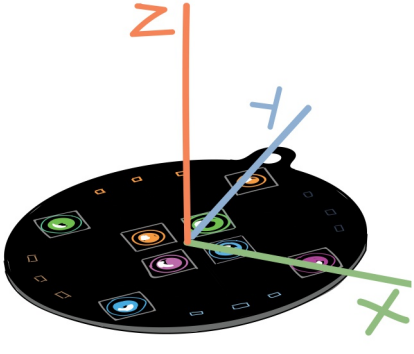
In the `setup` block, we start by telling the device to send messages at the right connection speed (the rate that information is transferred between your SpinWheel and your computer) using `Serial.begin(9600)`, Then, after gathering information from the SpinWheel's motion sensor, we can print the message we want (in this case the rotation in each dimension) using the `Serial.println()` function. Notice how the value in Serial Monitor changes as you spin the SpinWheel.

When you upload this sketch, you may notice that some of the small LEDs flash: this happens because the serial connection disturbs some of the electrical signals going to the LEDs. You can open this file from **Examples → ...?????** → ?????

```
#include "SpinWearables.h"
using namespace SpinWearables;

void setup() {
  SpinWheel.begin();
  // The next line ensures that the communication hardware
  // on our device is ready to send messages.
  // The name "Serial" is such for historical reasons
  // (it is the name for this type of communication).
  Serial.begin(9600); // The 9600 is the speed of the connection.
}

void loop() {
  // This line gets the information from the SpinWheel's
  // motion sensor.
  SpinWheel.readIMU();
  // Send a message to the connected computer.
  // The message will just be the value of the SpinWheel's
  // rotation around the x-axis.
  Serial.println(SpinWheel.gx);
  // Wait for 500 milliseconds (half a second) before you
  // start the loop function again.
  delay(500);
}
```

If you're curious, the sensor reading represents the number of degrees the SpinWheel rotates a second. Don't worry if this doesn't mean anything to you yet, we will explain it further in future adventures. To learn more about this and rotation around axes, check out the *Dancing with Color* adventure online!

For now, this output might make more sense if you look at it as a graph instead of just a list of numbers. To see a graph, close the Serial Monitor and navigate to **Tools → Serial Plotter**.

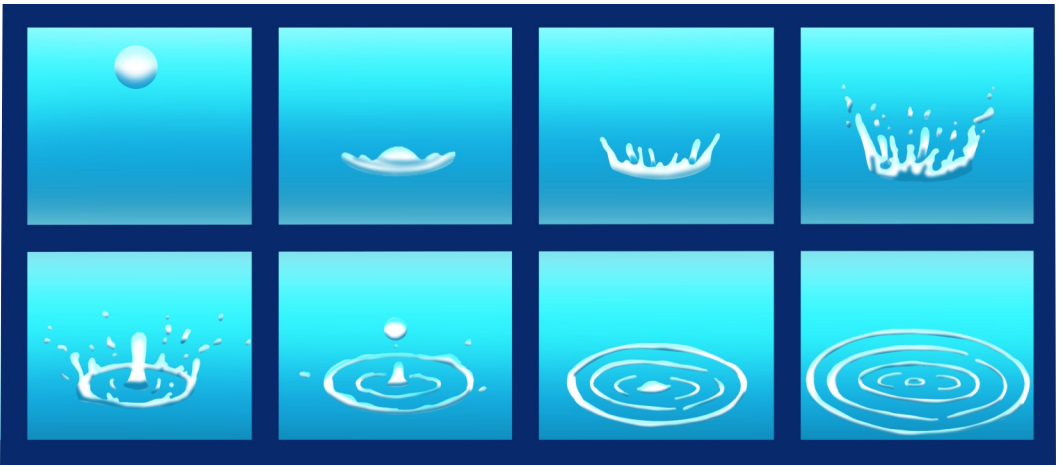


Graph of Serial Monitor output

This chapter expanded on some concepts that you've already seen and will be useful as you begin to code the SpinWheel in more complicated ways. Using the output from the motion sensor will be essential for making your own step counter, for instance. We will refer back to this guide in upcoming lessons and adventures and hope that you will use it as a reference.

::: further-reading

The Arduino community has very detailed resources on the programming language that we are using. You can start with their tutorial at www.arduino.cc/en/Tutorial/Foundations.



Creating Animations with the SpinWheel

An animation is created by rapidly cycling through a series of still images. You may be familiar with animations from watching cartoons, but we can also create animations with the SpinWheel's LEDs. After finishing this chapter, check out the *Intro to Animation* adventure on our website.

To create an animation or a video, we need a rapid sequence of still images, called frames. These frames must be displayed rapidly enough that they look like a smoothly changing pattern to our eyes. This is true both for a computer screen playing a video and for the SpinWheel lights changing their patterns.

In order to produce more dynamic patterns on our SpinWheel, we have to modify the loop block. Remember from the *Arduino 101* chapter that the loop block continuously repeats. So far, we have created patterns that produce the same result each time the loop is run. To begin writing more complex code, we need to introduce the idea of **variables**. Variables allow us to store information in the program and change that information

as needed.

To **define** a new variable we can add the following line to the loop block:

```
int which_LED = 1;
```

This line of code reserves a location in the memory of the computer, let's refer to that location by the name `which_LED`, and stores the value `1` there. By itself, this variable doesn't impact the output of the SpinWheel, but we can use it to store the location of the LED we want to light up.

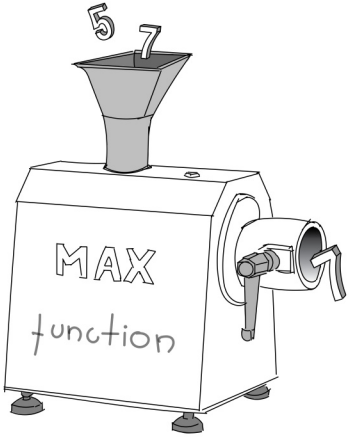
```
#include "SpinWearables.h"
using namespace SpinWearables;

void setup() {
  SpinWheel.begin();
}

void loop() {
  int which_LED = 1;
  SpinWheel.setLargeLED(which_LED, 255, 0, 0);
  SpinWheel.drawFrame();
}
```

Load this code from **Examples → ???????** and upload it to your SpinWheel. If you change the value of `which_LED`, you'll see a different LED light up.

To learn more about variables and other important concepts for creating programs, check out our *Programming Building Blocks* lesson. We encourage you to go back and forth between these pages as you deepen your understanding of programming the SpinWheel.

In this code, every loop still produces the same result; `which_LED` has the same value (of 1) every time `loop()` is run. Let's modify the code to change the selected LED every time loop is run. Load this file from **Examples → ???????** and look carefully at the comments to learn what each line does.

When you run this code, every loop increases the value of `which_LED` by 1. This value is then used in `SpinWheel.setLargeLED()` to indicate which LED to light up. This concept is further illustrated below. As each `loop()` block is run, the value stored in memory for `which_LED` is changed. Because the `loop` block runs many times a second, we have added the line `delay(500)` to pause the code for 500 milliseconds (0.5 seconds) before finishing the loop. Without the delay, the lights would change too quickly for us to see the change. Note that the small LEDs don't work when the delay function is used.



Something showing loop animation. Will need a larger caption as well.

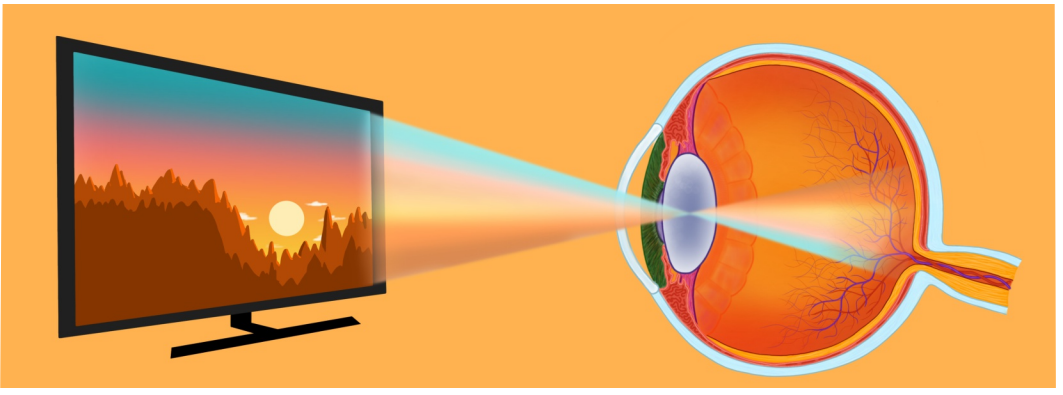
If you are interested in seeing how the value of `which_LED` changes as the code is run you can make use of the `Serial Monitor` introduced in the *Arduino 101* chapter. In the sketch you uploaded, there are a few commands that refer to Serial. In the `loop` block, the line `Serial.println(which_LED` allows us to see the value of `which_LED`. Open the Serial monitor (**Tools → Serial Monitor**) and then upload the sketch again to watch

`which_LED` increase as the code runs.

You may have noticed that after a few seconds your SpinWheel stops lighting up. This is because the value in `which_LED` no longer corresponds to one of the 8 LEDs on the SpinWheel. For example, if `which_LED` equals 10, the code will try to turn on LED number 10. Since it doesn't exist, nothing happens. To light up the SpinWheel again, unplug it and toggle the BAT/USB switch. This restarts the code, running `setup` again and starting `which_LED` over at 0.

What if instead you wanted this pattern to repeat itself over and over again? Check out the *Intro to Animation* adventure online to learn how to create computer generated animations with the SpinWheel!

What if the computer itself created the images? This is called generative art and you'll learn how to do this in the animations adventures that we have created and are found in our online resources.



Coding Building Blocks

This page will introduce you to many of the typical building blocks programmers use to write code. Knowledge of these concepts will help you write your own more complex code for the SpinWheel. Refer back to this content as you go through the online adventures at *spinwearables.com/book*.

Writing a computer program, whether an interactive website or the hidden brain of a robot, starts by writing a sequence of instructions in one of the many available computer languages. In this lesson, we will use **C++**, a very popular older language that runs well on simple computers like the SpinWheel's microcontroller. While **C++** has a specific set of commands and rules, the main ideas are common to other programming languages. In fact, the vast majority of computer languages share the same patterns, just like how many human languages share ideas such as the distinction between a noun and a verb, or the difference between a word and a sentence. We will describe the most important such patterns in this chapter.

Variables

Computer programs do one thing and one thing only: process information. That information can be the time of a mouse click, a voicemail on your phone, or a picture of the road taken by a self-driving car. Before processing such data, we have to tell the computer to store it in its memory. This is done using **variables**.

We will only discuss two types of variables: integers and decimals. Other types do exist, but we won't cover them here. If you store a number as an integer, it must be a whole number (like 2 or 3011) and cannot have a

Variables are like labeled shelves for information. When you need to save a number for later use, you put it in a variable of your choice. Above we have number 3 stored in the variable **a**, number 4 stored in variable **b** and the number 7 is about to be stored in a variable named **e**. We can pick the names for the variables ourselves (any sequence of letters that is not interrupted by whitespaces).

decimal component. On the other hand, a decimal can have a decimal part (like 0.4, 560.17, or 2.0). Integers are easier for a computer to work with because it does not need to store all of the data after the decimal point. Treating them separately from decimals lets us have faster code, which is especially important for small computers that don't have much storage space like in the SpinWheel.

To **define** a new integer variable you need the following line in your code:

```
int my_special_integer = 6;
```

This reserves a location in the memory of the computer, lets us refer to that location by the name `my_special_integer`, and stores the value **6** there. We can name the variable anything as long as it is a single word (or multiple words separated with underscores). We usually pick names that tell us something about the purpose of the variable. In this variable type, we can store any integer we want as long as it is not too large (no larger than roughly 30 000, due to limitations of how this computer stores integers).

If we want to work with decimals, we use the variable type `float` instead of `int`. The name comes from the early history of computers and is unimportant for our purposes (how the decimal point can "float" between the digits).

Here we stored an approximation of the number **π** in a variable with the name `pi`. We could then use this variable in other parts of our code to do computations that involve circles.

```
float pi = 3.1415;
```

Notice that throughout all of our code we have used the equality sign `=` to mean "store the value on the right in the memory cell on the left". This differs from the usual mathematical meaning of the sign, which usually means

"check if the left and right side have the same value". You can blame early computer scientists and their laziness for the misuse of this sign in most modern programming languages.

Functions

In computer programming, functions are commands that take a few variables and do something useful with them. Functions are reusable pieces of code. A function can act like a calculator, computing a new value based on the variables that are given to it. A function can also do something that affects the world around it, like blinking an LED, playing a sound, or sending a message.

Most programming languages have some functions built into them, similar to how a new cellphone comes with pre-installed apps. We can use these functions without having to write them ourselves.

Here is some code that uses an example function called `max` that takes two numbers as input and returns the larger number. The input values are also called the **arguments** of the function.

```
int number_a = 5;
int number_b = 7;
int resulting_number = max(number_a, number_b);
```

Let's step through each part of this code.

– `int number_a = 5` assigns the value of 5 to the integer variable `number_a`

– `int number_b = 7` assigns the value of 7 to the integer variable `number_b`

– `int resulting_number = max(...)` store the result of the function `max(...)` in the integer variable `resulting number`

– `max(number_a, number_b)` calls, or uses, the function `max(...)` with two arguments, `number_a` and `number_b`, and

returns the larger number.

The value stored in `resulting_number` in this case would be **7**.

Here is another example where one of the arguments for our function is specified directly, without first being stored in a variable. In this case, the value stored in `resulting_number` will be **8**:

```
int my_number = 6;
int resulting_number = max(my_number, 8);
```

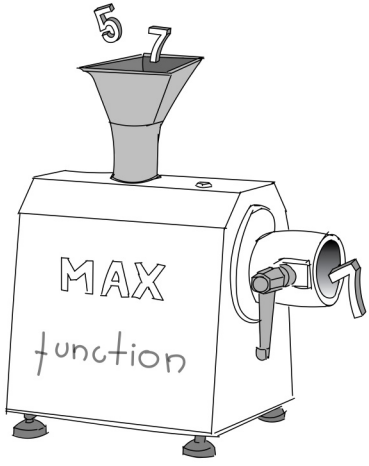
As you have seen, the typical syntax rules for the use of a function are to put its arguments inside parentheses immediately after the name of the function. You might have seen this in math class with trigonometric functions like `sin(x)` or `cos(x)`.

We can nest functions and use arithmetic operations on the arguments as well. For instance, here we will use two more functions, `min` which returns the smaller of two numbers and `sqrt` which returns the square root of a given number. Can you explain why the value stored in `resulting_number` in the following example is **4**?

```
int number_a = 5;
int number_b = 7;
int resulting_number = max(sqrt(number_a-1) * 2, min(number_b, 2));
```

Here is a hint: `sqrt(5-1) = sqrt(4) = 2`.

In this diagram, you can see the order in which the computer evaluates this piece of code. It first subtracts 1 from `number_a`, then evaluates the functions `sqrt()` and `min()`, next multiplies the output of `sqrt()` by 2, and finally evaluates `max()`. This is similar to how when solving a math equation, you first do anything within parentheses and then work your way out.



Functions are tools provided in a given programming language that are capable of ingesting a number of parameters and producing (a.k.a "returning") some new value that depends on the input parameters.

Creating your own functions

A large part of programming is creating your own functions and building interesting, complex, and useful functions out of small simple functions. Here we give an example of how to write your own function that takes two numbers, `x` and `y`, and returns their average, `(x+y)÷2`. We will name the function `avg`. Let us first write an example of how this function would be used if it already existed:

```
float x = 3.5;
float y = 2.5;
float resulting_number = avg(x, y);
```

In this code example, `resulting_number` will have the value of **3.0**.

To define this new function, we need to write down its name, together with the type of data it will be producing, followed by a set of computational instructions:

```
float avg(float first_argument, float second_argument) {
  return 0.5*(first_argument+second_argument);
}
```

Let's step through each part of this code.

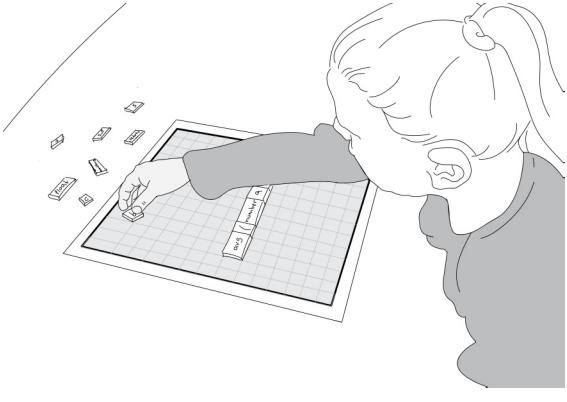
–`float avg(.....)`: The very first `float` specifies the type of data the function will produce (in this case they are decimals). This is followed by the name we have picked for our function, `avg`.

– `(float first_argument, float second_argument)`: In parentheses, we have a list of the arguments the function will be taking. Unlike when we call the function, we have to specify their types, so we wrote `float` to denote working with decimals. We also gave temporary names for these arguments so that we can refer to them in the function.

–`{ ... }` : The curly brackets surround the instructions of our function.

–`0.5*(first_argument+second_argument)` : This where the math happens in our function. It is simply the sum of the two arguments multiplied by one-half.

–`return`: a keyword to state that the result of this line of code should be returned to the code that called the function.



Now we have the pieces to begin to create our own functions. In the beginning, this may be like creating a poem using a limited set of magnetic word tiles. With practice, your bank of pieces will increase in size, enabling you to write more complicated code.

We can have multiple sequential instructions inside the block when the computation is more difficult. That is the purpose of the curly brackets `{ }` - to separate all the code that defines our function from the rest of the program that might be in the same file. For instance, here we will show how to compute the fourth root of a number. (The fourth root of a number, or x raised to the 1/4 power, can be computed by taking the square root of the square root and we will use this fact to write the fourth root function below.)

```
float root4(float x) {
  float intermediate_value = sqrt(x);
  return sqrt(intermediate_value);
}
```

Maybe the functions `avg` and `root4` seem too redundant to you, and you would prefer to always write `(x+y)/2` instead of `avg(x,y)`. This is a quite valid feeling for such short functions, but as you build more complex programs you will have longer pieces of code that would be cumbersome to repeat every time. Functions let you have a shorthand notation, so you do not need to make such repetitions.

Functions that do not return values

Functions can also be used to change something in the environment of the device instead of being used as advanced calculators. Such functions do not return a value and don't need a variable to store their output. One example is the `delay` function that pauses the computer for the length of time specified by the input variable. In the following example, calling the `delay` function will pause the program for 1000 milliseconds (which equals one second) before executing the next line:

```
a = calculation1();
delay(1000);
b = calculation2();
```

While this built-in function is nice, what if we want to specify the delay in seconds instead of milliseconds? When writing our own functions that do not have a return value, we specify the type of data that the function will be returning as `void`. This denotes that the returned value is empty or "void". Our new function takes the number of seconds as its input, calculates the number of milliseconds corresponding to the provided number of seconds and then uses the `delay` function to pause the program for that length of time. We do not need to use the `return` keyword because our function doesn't return a value.

```
void delay_seconds(int number_of_seconds) {
  int number_of_milliseconds = 1000 * number_of_seconds;
  delay(number_of_milliseconds);
}
```

Putting it all together

After we have created all the variables and functions we need for our code to do what we want it to do, we need to actually start the program. To do this, the program needs to know what function to run first. In different languages this is done differently. In our particular case, we do it by defining two special functions: `setup` and `loop`. Our computer is instructed to run these functions first. It finds the `setup` function and runs it before anything else. Usually this function is used to **set up** any settings we need in advance. Then the computer repeatedly runs the `loop` function, which is named this way because it **runs in a loop** (or repeats itself).

Let's look at a large example that includes all these features found at **Examples → SpinWearables → Paper Something → Counter**. It will use the `Serial.println()` function introduced in the *Arduino 101* chapter in order to send messages to the computer. Use the **Serial Monitor** tool in the Arduino software in order to see these

messages being sent back over the USB cable (**Tools → Serial Monitor**).

Read the comments in the code to try to understand what each line does. One great way to test your understanding is to consider what would happen if you change the code. For example, if you changed line **????????** to the line below, how would the output change?

```
counter = counter + 2;
```

Variables and functions are two of the most important building blocks of programming. As you go through the rest of online materials, you will be introduced to some other essential programming concepts.

SpinWheel Functions Reference

This chapter lists some of the most important functions available with the SpinWheel code library. With these functions you can detect motion and set the LEDs to different color patterns. This will serve as a reference as you code the SpinWheel in whatever designs you imagine.

LED manipulation

Turn on LEDs with drawFrame()

```
SpinWheel.drawFrame();
```

Below we will introduce a number of functions that let you control the LEDs. However, for the LEDs to change in brightness to reflect these functions, you also need to call `drawFrame()` when you are done. `drawFrame` is the only "non-instantaneous" function (commonly called "blocking" functions). It takes roughly 20ms for it to finish, during which time it uses a persistent of vision method to rapidly flash the small LEDs as necessary to mimic the required color. This lends itself to making 50 frames-per-second animations. By using `drawFrame()` you can make multiple modifications, preparing the final image, without the intermediate unfinished images showing up.

Turn off all LEDs with clearALLLEDs

```
SpinWheel.clearALLLEDs();
```

Use this function to reset the image, and set all LEDs to be dark, deleting any color information that they were previously set to.

Turn on all large LEDs with setLargeLEDsUniform

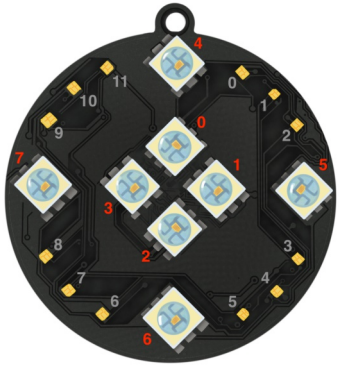
```
SpinWheel.setLargeLEDsUniform(255, 0, 155);
```

`setLargeLEDsUniform` takes three arguments, the red, green, and blue components of the desired color. Their range is between 0 and 255 (one byte).

Turn on all small LEDs with setSmallLEDsUniform

```
SpinWheel.setSmallLEDsUniform(255, 255, 0);
```

`setSmallLEDsUniform`, same as `setLargeLEDsUniform`, takes three arguments, the red, green, and blue components of the desired color. Their range is between 0 and 255 (one byte).



To control specific LEDs in the commands presented below use the numbering illustrated here.

Control a specific large LED with setLargeLED

```
SpinWheel.setLargeLED(4, 255, 155, 0);
```

The first argument of `setLargeLED` is a number between 0 and 7, denoting which of the 8 LEDs is to be turned on. See the above diagram for the numbering system. The other three arguments are the red, green, and blue components of the color.

Control a specific small LED with setSmallLED

```
SpinWheel.setSmallLED(4, 155, 255, 0);
```

The first argument of `setSmallLED` is a number between 0 and 11, denoting which of the 12 LEDs is to be turned on. The other three arguments are the red, green, and blue components of the color.

Turn on a range of large LEDs with setLargeLEDs

```
SpinWheel.setLargeLEDs(0, 3, 255, 0, 255);
```

The first and second arguments of `setLargeLEDs` are numbers between 0 and 7, denoting what range of the 8 LEDs is to be turned on. The other three arguments are the red, green, and blue components of the color.

Turn on a range of small LEDs with setSmallLEDs

```
SpinWheel.setSmallLEDs(2, 7, 255, 255, 0);
```

The first and second arguments of `setSmallLEDs` are numbers between 0 and 11, denoting what range of the 12 LEDs is to be turned on. The other three arguments are the red, green, and blue.

Set color using a single hex variable

```
SpinWheel.setLargeLEDs(2, 6, 0xffff00);
```

All of the above functions can be used with a single color variable in the standard hex notation (a different method of referring to colors).

Set overall brightness with setBrightness

```
SpinWheel.setBrightness(50);
```

This function takes only one argument, between 0 and 255, that sets the brightness of the LEDs. At maximal settings the large LEDs are blindingly bright and pull a total current of 480 mAh, which would deplete our battery in less than 10 minutes. Using the maximal setting would

cause the battery to wear out much quicker and would cause significant heating.

Custom LED patterns

Thanks to these functions you can easily set all of the LEDs to a preset pattern. They are useful building blocks for commonly used patterns that do not require significant customization.

Rainbow pattern with setSmallLEDsRainbow

```
SpinWheel.setSmallLEDsRainbow(120);
```

This function takes only one argument, between 0 and 255, that determines where a rainbow pattern in the small LEDs starts. For instance, if you use `setSmallLEDsRainbow(0)`, the start of the rainbow will be at small LED 0. To start the rainbow a half turn around the SpinWheel, instead use `setSmallLEDsRainbow(120)`.

Circular progress bar with setSmallLEDsProgress

```
SpinWheel.setSmallLEDsProgress(200, 0, 0, 255);
```

You can use this to create an arc around the SpinWheel with the small LEDs. It has four arguments: an angle that specifies the extent of the arc, and the red, green, and blue LEDs brightness. Each goes from 0 to 255. To have an arc that lights up LEDs 0-9 in blue, use `setSmallLEDsProgress(200, 0, 0, 255)`. You can also provide only two arguments, the angle and an rgb color.

Circular pointer with setSmallLEDsPointer

```
SpinWheel.setSmallLEDsPointer(100, 255, 255, 0);
```

This is similar to the above function, but instead you can specify the middle of the arc. To use our default arc length (check out our annotated source code), simply specify the angle for the middle of the arc (0-255) and the color

(either the red, green, and blue components individually or as rgb).

Color and brightness helpers

Various functions that help prepare colors or map a given number to a continuously changing brightness. A good place to see why these functions are useful is the set of lessons on animations.

A colorWheel

```
SpinWheel.setLargeLEDsUniform(colorWheel(100));
```

The `colorWheel` function turns a single number representing an angle into a color from the color wheel.

Here is an illustration of the triangular wave function.

The triangularWave function creates a pulsing pattern

```
SpinWheel.setLargeLEDsUniform(0,
triangularWave(150), 0);
```

The `triangularWave` function provides a convenient periodic function, useful in animations.

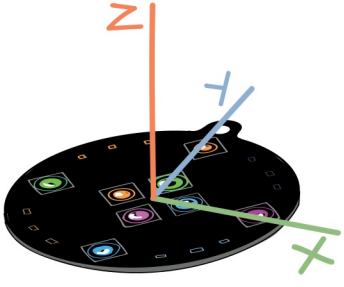
The parabolaWave function creates another pulsing pattern

```
SpinWheel.setSmallLEDsUniform(0, parabolaWave(155), 0);
```

Similar to the above function, the `parabolaWave` function creates a different periodic function based on the profile below.

Motion sensing and compass with readIMU

To request measurements from the sensor you need to call `readIMU`. Then the measurement data will be available in `ax`, `ay`, `az`, `gx`, `gy`, `gz`, `mx`, `my`, `mz`. The acceleration along the three axes `ax`, `ay`, `az` is measured in units of **g**, i.e. the [standard gravitational acceleration at Earth's surface]. The angular velocity components `gx`, `gy`, `gz` are measured in units of degrees per second. Lastly, the magnetic field `mx`, `my`, `mz` is measured in units of µT. There is also a temperature sensor, but we do not yet provide direct access to it.



The axes orientations of the SpinWheel motion sensor.

The *Dancing with Color* adventure which you can find online gives ideas of how the sensor can be used.