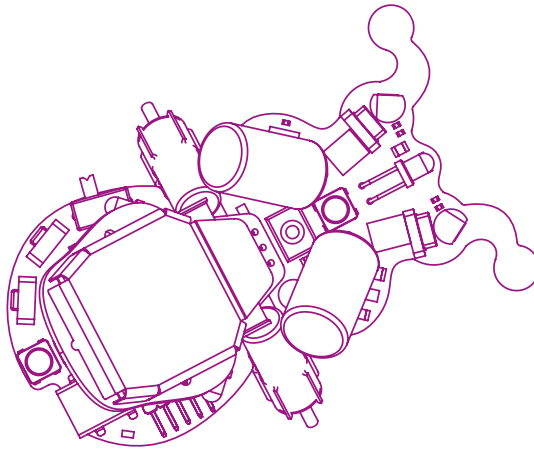




Ringo Educational Guide



Overview



Thank you!!

Thank you for your support in purchasing a Ringo robot. Ringo began as a project I started with my son and daughter. We wanted to design a robot together as a fun family project and only planned to make a couple of them. The idea evolved into a Kickstarter campaign which became very successful. We have been overwhelmed with the incredible community support and encouragement we received along the way. We are very hopeful to continue designing and producing many more robots in the future.

Your Ringo robot was assembled with love and care near Portland, Oregon. We had most of the components sent from over seas, but we actually populate all the parts on the board, as well as program and test right here in our own little shop. Most of the assembly was performed by students we hired from our local Clark Community College. We thank you for supporting a small business run by people of the Maker movement, and we hope Ringo will provide you a fun learning experience and many hours of entertainment. Cheers!

Acknowledgements

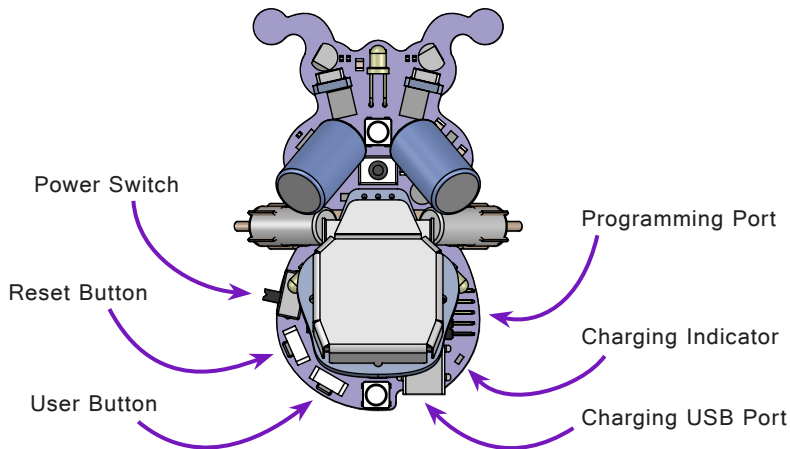
This guide will give you a quick orientation of the Ringo robot and how to program him, but first we want to give a few acknowledgements. Ringo is Open Source Hardware, which means his design is open to study and learn from. He wouldn't have been possible without others before us sharing their own open source creations with the world.

We have produced several of Ringo's primary systems based on open designs from SparkFun and AdaFruit, and Ringo's brain and programming environment were brought to the world by Massimo Banzi and the other fine people behind the Arduino project. Arduino has grown to become one of the most popular open source development platforms in history thanks to the wonderful user community which sprang up around the platform. Ringo's motor design was inspired by the Solar Popper robot produced by Solarbotics. Solarbotics is known for producing BEAM robotics kits, which are robots that carry out rather complex behaviors yet are based on very simple analog circuitry. Robotics physicist Mark Tilden is largely credited for the creation of the BEAM robot movement. I guess what we're saying is that creations like Ringo are possible thanks to the open sharing of knowledge between creative people. These fine people and many more we haven't mentioned have their fingerprints on Ringo as much as the people at Plum Geek.

We hope you will join us and continue in this spirit as you write interesting software and create behaviors for Ringo that can be shared and built upon in the community. We're very excited to have you along for the ride!

1
**SKILL
LEVEL**

Overview



Overview of Ringo's Parts:

Power Switch: This turns Ringo on and off. Click it forward to turn Ringo ON, click it rearward to turn Ringo OFF.

Battery: Ringo contains a 240 mAh Lithium Polymer battery, sometimes called a "LiPo" battery. LiPo batteries are awesome for a few reasons, but they do have a down side. If the battery is discharged too far, it becomes permanently damaged. Lucky for you, Ringo has a part onboard called a "voltage monitor". It's a little part that looks at battery voltage all the time, and if that voltage drops below 3.0 volts, it will immediately turn Ringo off, and will allow him to turn back on once the voltage is increased by charging the battery. All this to say, you don't have to worry about discharging Ringo too far. He will take care of himself if he is left on or run too long without a charge. If you notice Ringo is tending to reset on his own, it is likely because the battery voltage is getting low. Turn him off and plug him into a charging source for a while and he'll be happy again.

Charging Ringo: Ringo has a self contained battery charging circuit. Whenever he is plugged into a power source via either the Charging USB Port or the Programming Port, he will automatically charge his battery. This happens whether the Power Switch is turned on or off. It is okay to run Ringo while charging. The charger will automatically turn off when Ringo's battery is full (about 4.2 volts), so you cannot over charge him. Just plug him in and let him handle it. Whenever Ringo is charging, the RED Charging Indicator will be lit.

Overview

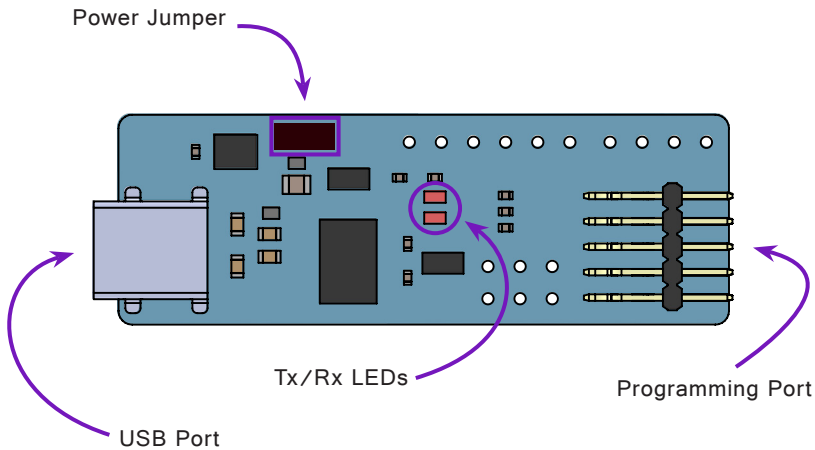
Charging USB Port: The USB port present on Ringo does not allow for programming him. However, it is useful for charging his battery. USB cables provide a 5 volt power source which is used for charging. Feel free to plug him into a USB power source via this port for charging without the programming adaptor. (When the Programming Adaptor is plugged into a USB power source, it will pass this 5 volt power along the programming cable to Ringo's Programming Port, and Ringo will use this source whenever available to automatically begin charging his battery).

Reset Button: This button connects to a special pin on Ringo's brain. Whenever pressed, it will halt Ringo's processing, and when released, Ringo will startup fresh from the beginning of his program. This button does the same thing as turning the power switch off then back on.

User Button: This is a button you can use for your own purposes. Ringo can be programmed to look at this button and determine if it is being pressed, and if it is being pressed, to carry out some action, to change his behavior in some way, whatever you can think of. We'll cover the use of this button later in this guide.

Programming Port: This port connects Ringo to the Plum Geek Programming Adaptor. It is used to load Ringo with his programming. It can also charge Ringo's battery when the Programming Adaptor is plugged into a USB power source.

Overview



The Programming Adaptor is used to communicate between your computer and Ringo's brain. We'll go into more detail later in the installation section, but wanted to provide a quick overview of the programmer here.

USB Port: Connect this to your computer. This requires a USB A to Mini-B cable.

Programming Port: This connects to Ringo via the programming cable.

Tx/Rx LEDs: These LEDs blink whenever data travels between Ringo and your computer through the programmer.

Power Jumper: This jumper connects the power supplied by your computer via the USB Port to the Programming Port, which allows Ringo to charge through the Programming Adaptor. Normally this should be left in place. It is however occasionally useful to remove this jumper. As the battery in Ringo reaches its maximum charge level, the charging circuit will continue to cycle on and off while Ringo is running, which causes minor voltage fluctuations in Ringo's power system. This is okay and normal, except that while taking readings from Ringo's sensors (like the light sensors), you'll see these readings spiking a lot more while charging is allowed. In this case, it may be useful to prevent Ringo from charging while still allowing Ringo to communicate with your computer. Remove the jumper to suspend charging. If you do remove the jumper, place it back on one of the pins so the little jumper connector is not lost on your desk.

Software Installation



Install the Arduino IDE (Integrated Development Environment)



Access the Internet

In order to program your Ringo, you'll need to download the newest version of the Arduino software first from www.arduino.cc. (it's free!). This software, known as the Arduino IDE, will allow you to program your Ringo to do exactly what you want. It's like a word processor for writing programs. With an internet-capable computer, open up your favorite browser and type in the following URL into the address bar:

 arduino.cc/en/main/software

1

Download

Click on your appropriate computer operating system next to the “+” sign.

- + Windows
- + Mac OS X
- + Linux: 32 bit, 64 bit
- + source



 Choose the appropriate Operating System installation package for your computer.

1

**SKILL
LEVEL**

This section has been largely reproduced from the [SIK Guide](#) for the [SparkFun Inventor's Kit](#).

2

Visit www.plumgeek.com and follow the instructions to install the required driver for the programming adaptor. It is super important that you do this before connecting the programmer to your computer.

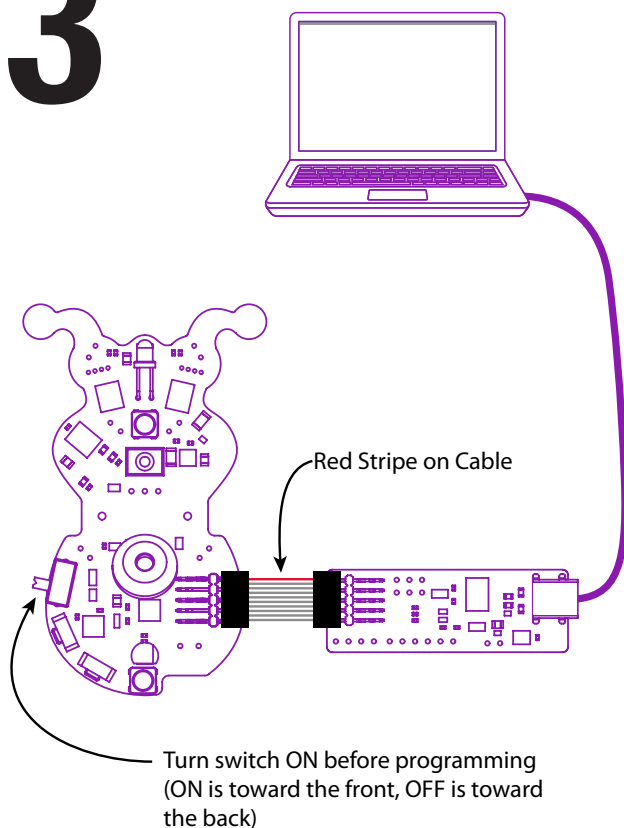


Software Installation

Connect a USB cable between your computer and the Plum Geek Programming Adaptor. Then connect the Programming Adaptor to your Ringo robot using the supplied cable.

You will require a USB A to Mini B cable (commonly supplied with digital cameras and the like).

3



1

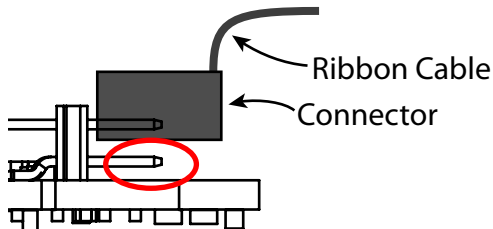
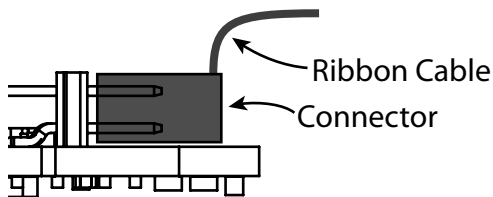
**SKILL
LEVEL**

This section has been largely reproduced from the [SIK Guide](#) for the [SparkFun Inventor's Kit](#).

Software Installation

Warning! It is rather easy to attach the programming connector to the top row of pins only which is incorrect. Please look closely as you are attaching the programming connector to be sure it engages both the upper and lower set of pins at the same time.

Warning! Take care to not bend the pins when handling Ringo or attaching the programming cable.



1
SKILL
LEVEL

This section has been largely reproduced from the [SIK Guide](#) for the [SparkFun Inventor's Kit](#).

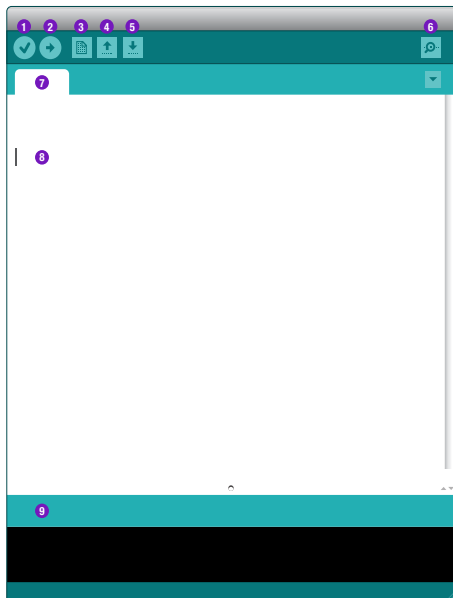
Software Installation

Open the Arduino IDE:



Open the Arduino IDE software on your computer. Look around and get to know the interface. We aren't going to code right away, this is just an introduction. This step is to set your IDE to identify the Plum Geek Programming Adaptor and the attached Ringo robot.

- 1 Verify:** Compiles and approves your code. It will catch errors in syntax (like missing semi-colons or parenthesis).
- 2 Upload:** Sends your code to the attached Ringo robot. When you click it, you should see the red lights on the Plum Geek Programming Adaptor blink rapid. IT IS VERY IMPORTANT TO MAKE SURE YOUR RINGO IS SWITCH ON PRIOR TO UPLOADING.
- 3 New:** This button opens up a new code window tab.
- 4 Open:** This button will let you open up an existing sketch.
- 5 Save:** This saves the currently active sketch. Note, the sketch is not saved automatically when uploading or verifying.
- 6 Serial Monitor:** This will open a window that display any serial information your Ringo is transmitting. It is useful for debugging.
- 7 Sketch Name Tab:** This shows the name of the sketch you are currently working on. You will often see extra tabs here also. The extra tabs represent extra documents that contain some of the under-lying code used by Ringo. Normally you won't edit these extra tabs. If you become a more advanced user we would encourage you to look around these tabs and feel free to modify. If you come up with better ways to write or organize these background functions please share with us. This is an area we would appreciate help with from the user community. Thanks!
- 8 Code Area:** This is the area where you compose the code for your sketch.
- 9 Message Area:** This is where the IDE tells you if there were any errors in your code, or if there are errors in the process of uploading your code to your Ringo robot.



These are the most important buttons in the IDE:



Verify



Upload



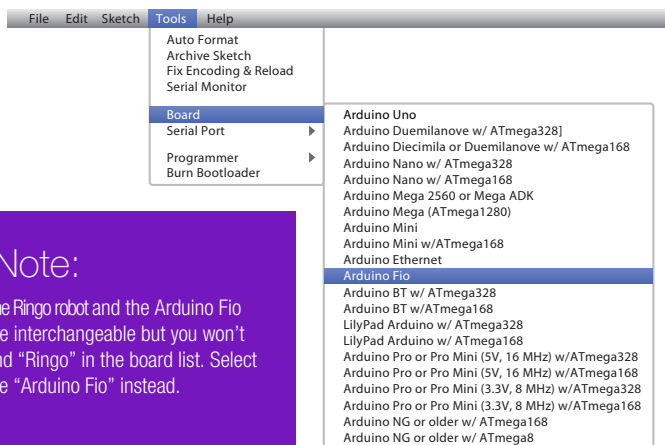
Serial Monitor

1
SKILL
LEVEL

This section has been largely reproduced from the [SIK Guide for the SparkFun Inventor's Kit](#).

Software Installation

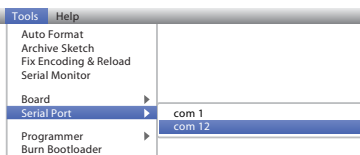
4



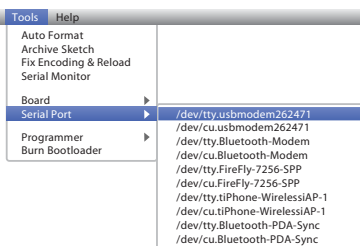
Select your Serial Device



Select the serial device of the Ringo robot from the Tools | Serial Port menu. This is likely COM3 or higher. It may take a few seconds for a new COM port to show up after connecting the PG Programmer. If a port doesn't show up, try disconnecting and re-connecting the programmer. Make sure you have installed the FTDI drivers in the earlier step.



Select the serial device of the Ringo robot from the Tools | Serial Port menu. On the Mac, this should be something with `/dev/tty.usbmodem` or `/dev/tty.usbserial` in it.



For Linux, visit the web page:
<http://www.arduino.cc/playground/Learning/Linux>

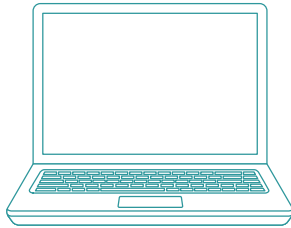
1

**SKILL
LEVEL**

*This section has been largely reproduced from the **SIK Guide for the SparkFun Inventor's Kit**.*

Software Installation

5



Visit the Plum Geek website



Visit the [plumgeek.com](http://www.plumgeek.com) website. Download libraries & examples, and install them in your Arduino folder.

In order to easily program and use your Ringo robot, you will require several library folders, as well as a folder full of examples. You will need to place the three library folders (**RingoMsTimer2**, **RingoWire**, and **Adafruit_NeoPixel**) into your **Arduino/libraries** folder. You will also want to place the **Ringo_Examples** folder into your **Arduino/examples** folder. Please see the illustrations on the following page.

Note: The Ringo libraries require Arduino IDE version 1.0.6 or later.

Note: The location of your Arduino Libraries directory may be slightly different than the path provided below. Follow the instructions for library installation on the Plum Geek website for known alternate directory structures you may encounter. The library install only needs to be done once.

1
SKILL
LEVEL

This section has been largely reproduced from the [SIK Guide](#) for the [SparkFun Inventor's Kit](#).

Software Installation



Unzip the file "Ringo_Libraries". It should be located in your browser's "Downloads" folder. Right clic the zipped folder and choose "unzip".

Start → Programs → arduino → libraries

Copy the following folders (contained inside the .zip file) into Arduino's folder named "libraries".

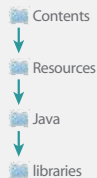
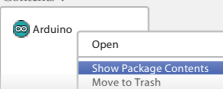
RingoMsTimer2
RingoWire
Adafruit_NeoPixel



Unzip the file "Ringo_Libraries". It should be located in your browser's "Downloads" folder. Right clic the zipped folder and choose "unzip".



Find "Arduino" in your applications folder. Right click(ctrl + click) on "Arduino". Select "Show Package Contents".



Copy the three folders RingoMsTimer2, RingoWire, and Adafruit_NeoPixel into Arduino's folder named "examples".



Unzip the file "Ringo_Examples". It should be located in your browser's "Downloads" folder. Right clic the zipped folder and choose "unzip".

Start → Programs → arduino → examples

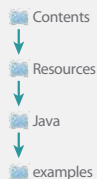
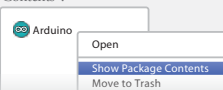
Copy the following folder "Ringo_Examples" into Arduino's folder named "examples".



Unzip the file "Ringo_Examples". It should be located in your browser's "Downloads" folder. Right clic the zipped folder and choose "unzip".

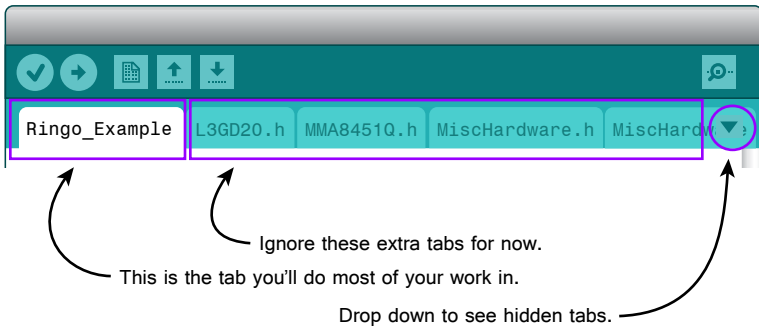


Find "Arduino" in your applications folder. Right click(ctrl + click) on "Arduino". Select "Show Package Contents".



Copy the three folder "Ringo_Examples" into Arduino's folder named "examples".

Software Installation



A quick note about tabs in the Arduino IDE...

Arduino likes to store all the files associated with a sketch in a single folder. It then opens the main sketch (the first tab) as well as any additional files it finds in the folder. The additional files are opened in additional tabs across the top of the IDE window.

Don't let these extra tabs intimidate you. For now you can largely ignore them. These extra tabs hide the background functions used by Ringo. Rather than install them as a library (in which case they wouldn't appear as tabs), we've included them with each sketch so more advanced users can easily study and edit them. As you are getting started, keep all your code in the first tab, which has the same name as the file name of the sketch you are working on.

Those more experienced with coding are encouraged to have a look at these extra tabs. Feel free to modify or improve them as the Ringo software is a work in progress and we welcome the user community to add features and make improvements. There are more tabs than are visible in the window (unless you really stretch out the window). To access the hidden tabs, use the drop down arrow circled above.



Let's cover a few basics...

I know you're itching to make Ringo do cool stuff, so play with some of the pre-loaded behaviors for a while. Eventually, you'll want to customize how Ringo behaves. For this, you'll need to learn a few basics about the code. This guide is not intended to be a complete course in programming. There are lots of great resources for this (see the end of this section for some links!) However, we'll try our best to start you off in the right direction. Let's dive right in.

Below you'll see a box containing code. We'll use boxes like this throughout the guide to represent code you'll write or edit in the Arduino IDE. The important parts of the code in a given example are highlighted in purple.

```
/*
  Opening Heading. This area includes basic information about the program.
  It is not actually loaded onto Ringo. These are just notes for the reader.
*/

#include "MiscHardware.h"    //this makes the IDE include some outside
                             //files when loading code into Ringo

void setup(){                //the setup() function configures Ringo on startup
  HardwareBegin();           //Sets all of Ringo's I/O pins correctly
  otherstuff();               //there may be other lines of code inside setup()
}                             //this closing curley bracket is the end
                             //of the setup() function

void loop(){                 //this is where the real action happens!
  commandsToDoStuff();        //functions inside the loop() function make Ringo
  doMoreCoolStuff();          //do all his cool stuff. You will write most of
  evenMoreFun();              //your code inside the loop() function
}                             //this closing curley bracket is the end
                             //of the loop() function
```

Code Basics

Code! Yay! Let's talk about the above example...

Each sketch includes code and comments. Comments are super important because they give human readers important notes about what's going on in the sketch. Comments are ignored by the Arduino IDE when it compiles the code before sending it to Ringo's brain. Comments can be written two ways.

A pair of forward slashes denotes everything else following them on the same line as a comment, and thus is ignored by the compiler.

```
thisIsCode(); //this part is a comment. This part is ignored by the compiler.
```

But what if your comment is going to require more than one line to write? There's a solution for that. You can begin a comment block with a forward slash followed by an asterisk. Then you can write as many lines of comment as you like. You then end the comment block with an asterisk followed by a forward slash. Here is an example of a comment block including the starting and ending characters.

```
/*  
This is a comment block. You can write as much as you want inside a  
code block and the comment can take up many lines. You do have to end  
a code block eventually with closing characters.  
*/
```

Great. Now let's talk about the `#include "somefile"` line. This tells the compiler to copy and paste the referenced file at this point in the code. It is then compiled in as if you had actually pasted it in this place. This is a good way to include some external files (which may be really long) at the top of your code without making your main code window look cluttered.

You'll learn more about include files eventually. For now, just leave the existing `#include` lines in the examples and be careful not to edit or delete them. If they are altered, the Arduino IDE may refuse to compile your code sketch.

Code Basics

Now lets talk about the `void setup()` function. Everything between the opening curly brace `{` and the closing curly brace `}` of this function is run one time when Ringo first powers up (and also after you press and release the Reset user button on Ringo's body). This function is useful for "setting up" Ringo's brain for operation. You should include a function inside `setup()` called `HardwareBegin()`; which sets up Ringo's brain to use the electronic parts included on Ringo's body.

The `setup()` function for many of the Ringo examples is shown below.

```
void setup(){
  HardwareBegin();           //initialize Ringo's brain to work with his circuitry
  SwitchButtonToPixels();    //set shared line to control NeoPixel lights
  PlayStartChirp();          //play startup chirp and blink eyes
  delay(1000);               //wait 1 second
  NavigationBegin();         //startup the gyro and accelerometer
  RestartTimer();            //capture start time
}
```

After `setup()` finishes, the `loop()` function runs over and over again, forever (or until you turn off or reset Ringo, or until his battery runs too low and he shuts off automatically). The `loop()` function is where the bulk of you program usually lives.

Further learning...

You'll learn a lot as you work your way though this guide which will show you how to use all of Ringo's parts on a basic level. However, this guide will not teach you all the nuances of programming. If you'd like to learn more on your own, please check out the following links.

Learn the basics about sketches and programming technique here:

<http://www.arduino.cc/en/Tutorial/Foundations>

Learn the core Arduino functions. This is the "go to" reference for functions:

<http://www.arduino.cc/en/Tutorial/HomePage>

(continued on next page)

Code Basics

If you look around on YouTube you'll find lots of great explanations as well as awesome projects people around the world have created with Arduino. Here are a few video courses we think you'll enjoy. Remember that Ringo is basically an Arduino board without connecting headers - we've just connected the electronics directly to the microcontroller, so many of the examples you'll find online are still applicable to Ringo.

Arduino Course for Absolute Beginners

<https://youtu.be/QNTaQ5qjniE?list=PLYutciIGBqC2rqIBw3wVX4LjFlcjtWjGP>

Tutorial Series for Arduino by Jeremy Blum

https://youtu.be/fCxzA9_kg6s?list=PLA567CE235D39FA84

If you come across any other great resources you feel would be helpful to others, please send us a link and we'll add them to this guide.

Using Ringo's Color Lights

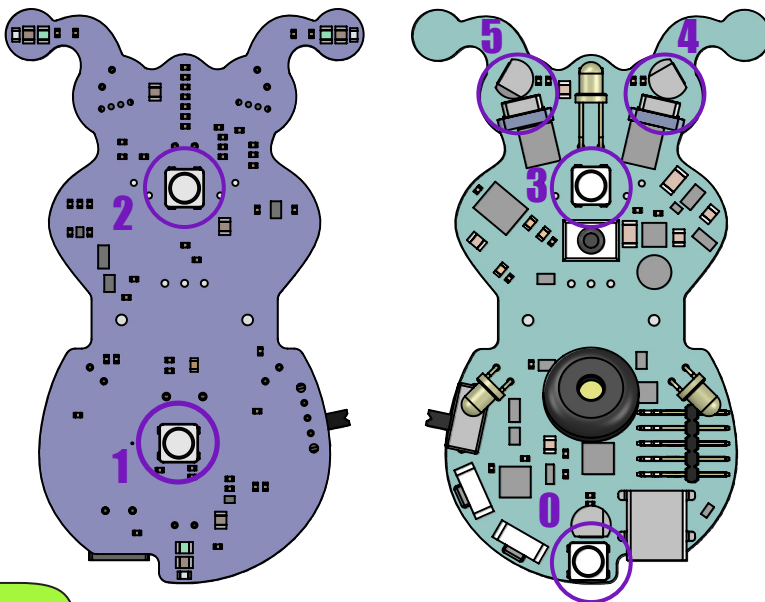


Ringo is fitted with six color lights. They are called "RGB" lights, which stands for "Red, Green, and Blue". Each light actually contains three small colored lights inside. You guess it - they are red, green, and blue. It turns out you can create any color you can think of by mixing different amounts of these three primary colors. Your TV screen creates colors in the same way.

The RGB Lights in Ringo were made popular by Adafruit Industries, coining the name "NeoPixels". We owe them a huge high-five for writing the code required to communicate with these awesome little lights.

Each light can have each of its red, green, and blue LEDs set to any level between 0 and 255. When set to 0, the given color is off, and when set to 255, the given color is on at max brightness. The NeoPixel lights are extremely bright, so a value from 100 to 150 is still plenty bright.

As stated, Ringo has six of these color lights. Computers like to number things beginning with 0. So the lights are actually numbered 0 through 5 (not 1 through 6 as you may expect). This address number of each light is printed next to it on the circuit board for easy reference. Check out the graphics below showing each light with its address.



1
SKILL
LEVEL

Using Ringo's Color Lights

A few setup steps...

You need a few lines of code in your project to tell the Arduino IDE to compile in the super special Adafruit NeoPixel driver software. This code is already present in the example projects, so don't let this part scare you. The code belongs in your main project page above the line: `void setup(){`

Like this:

```
#include "MiscHardware.h"
```

This include line will insert the `MiscHardware.h` file, which itself includes a special line: `#include <Adafruit_NeoPixel.h>` By including `MiscHardware.h`, the IDE will automatically include the `Adafruit_NeoPixel.h` file which allows Ringo to talk to his lights.

You'll also need to add a line of code inside your `void setup(){` function as highlighted below. Place it right after the `HardwareBegin();` but still inside the closing curly brace.

```
void setup(){
  HardwareBegin();           //Sets all of Ringo's I/O pins correctly
  SwitchButtonToPixels();    //set shared line to control NeoPixel lights
  // (it's okay if there is more stuff below here)
}
```

The line that controls the NeoPixel lights is also shared with the User button. By running the function `SwitchButtonToPixels();` this shared line is configured for NeoPixel use.

Time for fun! Let's Do This!

Now we're ready to make the lights go! You will control the lights with the function `SetPixelRGB();` Here is an example of how the function is formatted.

```
SetPixelRGB(address,red,green,blue); //example formatting of function
```

You'll replace the word "address" with the address of the pixel you want to adjust. Remember this is a number between 0 and 5. Then you'll place a number between 0 and 255 in place of the words "red", "green", and "blue". The number you use will correspond to how bright or dim each of the three colors are.

Using Ringo's Color Lights

Lets see if we can make the light on the top of Ringo's body turn blue. You'll be writing most of your code inside the `void loop(){` function (at least until you start writing your own functions!). Make your `void loop()` function look like this:

```
void loop(){  
  SetPixelRGB( 3, 0, 0, 200); //set pixel 3 to 0 red, 0 green, and 200 blue  
  RefreshPixels();  
}
```

Now try to compile and load this code onto your Ringo. About a second after you turn on the power switch, the light on the front top of Ringo's body should be bright blue. Now let's talk about what happened, and why.

The `SetPixelRGB()` is taking four numbers, called "arguments" which control what the function does. The first number, in this case "3" tells Ringo's brain to target the pixel at address "3". The next three numbers tell the pixel at address 3 how bright to turn on the red, green, and blue lights inside the pixel. In this case, the red is set to 0 so it won't be on at all, as well as the green will be set to 0 so it won't be on either. The last number is 200, which corresponds to the blue led which will be turned on at a brightness of 200, which as you can see, is pretty bright!

You may have noticed I added one more line of code for you: `RefreshPixels();`. When the first line is executed (the `SetPixelRGB` line), Ringo's brain actually sends a signal over the circuit board to pixel number 3. Pixel number 3 is told what color it is eventually going to become, but it doesn't actually change to that color just yet. The pixel waits for a second signal which tells it to actually change to the commanded color. This is called "latching" the new color.

The line `RefreshPixels();` causes Ringo's brain to command all the pixels on the robot to actually latch their commanded color. This is the instant when the lights actually change color. This may sound like a needless extra step, but it's actually a super smart feature. If you were making a large video screen (like the jumbotron at a sports arena), a computer brain is able to update all pixels on the display at the exact same instant, which helps the video images to remain smooth and synchronized.

Using Ringo's Color Lights

Using the basic example above, we can start to play with the numbers in the function to change the behavior of the pixels. Let's say your favorite color is something different than red, or green, or blue. Like, PLUM! You can easily create different colors by mixing different amounts of the three primary colors Red, Green, and Blue. Try this example code. Compile it, load it, and observe what happens.

```
void loop(){
  SetPixelRGB( 3, 220, 30, 160); //set pixel 3 to 220 R, 30 G, and 160 B
  RefreshPixels();
}
```

Sweet! Purple, right? I think you're getting the idea now. So what if you want to control more than one pixel at a time? What if we want to make both of Ringo's eyes turn purple? We just do the same thing, except we call the `SetPixelRGB` function more than once. We call it once for the right eye, and again for the left eye, then finally after both eyes have been commanded to a new value, we will latch the new colors by executing the `RefreshPixels()` function. Like this:

```
void loop(){
  SetPixelRGB( 4, 220, 30, 160); //set pixel 4 (Right eye)
  SetPixelRGB( 5, 220, 30, 160); //set pixel 5 (Left eye)
  RefreshPixels();
}
```

Purple eyes? Easy right? I think you've got the hang of it now. Now lets have some real fun. One more example then you can go off and play on your own.

```
void loop(){
  SetPixelRGB( 4, 220, 30, 160); //set pixel 4 (Right eye)
  SetPixelRGB( 5, 220, 30, 160); //set pixel 5 (Left eye)
  RefreshPixels();
  delay(100);
  SetPixelRGB( 4, 30, 220, 210); //set pixel 4 (Right eye)
  SetPixelRGB( 5, 30, 220, 210); //set pixel 5 (Left eye)
  RefreshPixels();
  delay(100);
}
```

Using Ringo's Color Lights

After you take that example for a spin, see if you can figure out what's going on. And what's up with `delay(100);` I added to the code? Any idea why the main function we're editing is called `loop()` ? Does anything in this example appear to be looping? Think it over and discuss it. Then see the "Discussion Answers" at the end of this section and see if you've got the right idea.

Gotcha! A couple warnings to keep in mind...

In this section of each lesson, you'll find some helpful tips and warnings to keep in mind. Ringo is designed with robust components and he has some built-in protection systems to keep him safe. Here are some situations you may run into.

Over Current Limit:

Each NeoPixel LED draws a current of 60 milliamps when all three of its colors (red, green, and blue) are turned on full brightness. If you did this to all six NeoPixels at once, then Ringo would need to draw about 360 milliamps from his battery. Battery current draw is beyond the scope of this lesson, but suffice to say, Ringo's battery can only safely supply about 200 milliamps. For this reason, all the parts on Ringo that are capable of drawing a lot of current (the NeoPixels, motors, sound chirp, underside edge sensors, and the three infrared emitters) are all powered through a limit switch that limits current to the safe 200 milliamps. Ringo's brain is however not powered through this switch - so his brain will usually remain active even during current limit. When the current limit is engaged, you will see the pixels and other current limited parts stop working or behave in strange ways. This is okay and does not cause them any harm. Just edit your code and reduce the brightness of your NeoPixels, speed of your motors, etc. In most cases you'll never hit this current limit as you're not usually running motors full speed while turning on lots of NeoPixels at high brightness. As you saw in this lesson, even at only 100 brightness, the light is still pretty bright, so going full blast to 255 is usually just wasting current.

NeoPixel/Button Shared Line:

We based Ringo on the Arduino UNO as it is the most popular Arduino board. However, the processor on that board (and thus the processor in Ringo) is somewhat limited on how many signal lines it has. For this reason, we have doubled up on some signal lines.

Using Ringo's Color Lights

The signal line controlling the NeoPixel LEDs is also used to sense if someone is pushing the User Button. When controlling the NeoPixels, this line is an output. When looking at the button, it is an input. By default it is set to control the NeoPixel lights.

Pressing the User Button will block communication to the NeoPixel lights while it is pressed. This does not harm the button, the NeoPixels, or Ringo's brain, but it is something you should be aware of. We've provided two simple functions that can be used to change the mode of this line as follows:

```
SwitchPixelsToButton(); //use this function before reading button
SwitchButtonToPixels(); //use this function to resume pixel operation
```

The Motor Direction control lines are also doubled up with the debugging lines and must be switched in a similar way. See the motor control lesson for more information.

It no workey. (Sad Ringo).

If you spend more than five minutes learning how to code, you will find your patience tested. Writing code is really pretty simple, but there is a small catch. Computers expect to see instructions formatted a very specific way. There are different rules in each computer language that must be followed. A piece of software that runs in the background on your computer (called a "compiler") reads your code and turns it into a bunch of 1's and 0's that make sense to Ringo's brain. The it is very easy to make the compiler grumpy, in which case it will throw a fit and refuse to do what it's told. The solution to keeping it happy is to pay close attention to how you format your code. Here are some common mistakes you may be making, and how to correct them.

```
SetPixelRGB( 1, 120, 50, 90); //correct upper-lowercase function. This works.
setpixelRGB( 1, 120, 50, 90); //no workey
setpixelrgb( 1, 120, 50, 90); //no workey
SETPIXELRGB( 1, 120, 50, 90); //no workey

delay(100); //correct delay. This works.
Delay(100); //no workey
```

Using Ringo's Color Lights

Be sure you're using commas between the arguments.

```
SetPixelRGB( 1, 120, 50, 90); //using commas correctly. This works.
SetPixelRGB( 1; 120; 50; 90); //wrong. semi-colons between arguments. no workey
SetPixelRGB( 1 120 50 90); //forgot commas between arguments. no workey
```

Be sure you're using normal parentheses around arguments.

```
SetPixelRGB( 1, 120, 50, 90); //normal parenthesis. This works.
SetPixelRGB{ 1, 120, 50, 90}; //curley braces are wrong. no workey
SetPixelRGB[ 1, 120, 50, 90]; //brackets are wrong. no workey
```

Don't forget the semi-colon at the end of every line! (Yes, I realize there are some lines in the example code that don't have semi-colons at the end. The mean people who created the C language did this to confuse you. We'll discuss this later). For now, suffice to say that missing a semi-colon is the most sure way to upset the compiler.

```
SetPixelRGB( 1, 120, 50, 90); //correct semi-colon. This works.
SetPixelRGB( 1, 120, 50, 90) //oops. no semi-colon. upset compiler. no workey
SetPixelRGB( 1, 120, 50, 90), //this is a comma, not a semi-colon. no workey
```

As picky as the compiler can be, it is usually okay with extra blank spaces. Sometimes adding some extra space between arguments of a function can make the function easier to read.

```
SetPixelRGB(1,120,50,90); //the compiler sees this the same as...
SetPixelRGB( 1, 120, 50, 90); //this
```

Discussion Answers:

1) What's up with the `delay(100);` inserted into the code? Well, "delay" does just what you'd expect it to. It causes Ringo's brain go spin in a circle and do nothing for a certain amount of time. The argument passed is the number of milliseconds Ringo's brain should wait before proceeding. Keep in mind there are 1000 milliseconds in one second. So this line causes Ringo to pause for a tenth of a second before going ahead in the code.

2) Why is the main function we're editing called `loop()` ? As you probably noticed, Ringo keeps doing the same thing over and over. He sets his eyes to one color, then latches the color, then delays a short time, then sets his eyes to a different color, then latches the new color, then delays a short time... then he does it all again. The `void loop()` function runs all

Using Ringo's Color Lights

the code contained within its curly braces. Once it reaches the end, it “loops” and does it all again. This loop function is where you'll be writing most of your code for a while (you'll eventually start writing your own stand-alone functions). Forget about the word “void” at the start of the line for now. Just know that it needs to be there and you don't need to know why at this point.

Experiments & Challenges:

Now that you've mastered controlling Ringo's color lights, go ahead and experiment. Here are some suggestions and challenges.

- 1) What happens if you play with the argument inside the `delay();` function? What happens if you make the delays different values?
- 2) What happens if you use the `SetPixelRGB()` function to control pixels at different addresses? Can you make Ringo's bottom lights work without any further hints from me?
- 3) Good job! You made the bottom lights work! What happens if you now take Ringo into a dark room and place him on different surfaces while the bottom lights are lit up? (Hint, look for something semi-opaque, a clear table top, and a white surface and see what happens).
- 4) What color do you get if you make all three Red, Green, and Blue values the same?
- 5) What happens if you run this code:

```
void loop(){  
  SetPixelRGB( 4, 255, 255, 255); //set pixel 4 (Right eye)  
  SetPixelRGB( 5, 255, 255, 255); //set pixel 5 (Left eye)  
  RefreshPixels();  
}
```

Woah! That's BRIGHT!!! Now take Ringo into a dark room. What do you see projected on the walls of the room? Why is this happening?

- 6) Remember the example where we made Ringo's eyes flash between two colors? Can you make his eyes flash between three different colors now? What about ten colors? Twenty? Try lots of different mixes of color and different delays.
- 7) Write the code on your own to turn on any NeoPixel then wait a short delay. How do you now make it turn back off? Can you make a certain NeoPixel blink on and off repeatedly?

Using Ringo's Color Lights

8) Can you give Ringo oogley eyes? Try making a color alternate between the two eyes. (The right eye turns on one color while the left is not turned on, then you turn on the left eye to the same color while turning off the right eye - then loop).

9) Can you make Ringo look like his eyes are hypnotized?

10) Can you do something cool with Ringo's lights that we haven't even thought of yet?

Bonus Function:

Now that you're familiar with the `SetPixelRGB()` function, we'll tell you about another function you can use to control the lights.

```
SetAllPixelsRGB(red,green,blue); //example formatting of function
```

The `SetAllPixelsRGB()` function is useful for doing exactly what you would expect. This single function sets every light to the same value. It also automatically refreshes the pixels, so you don't need to call the `RefreshPixels()` after running it. This function is most useful for turning off all the NeoPixel lights at the same time. Occasionally, if Ringo is reset (or the User button is pressed) during a command to the lights, some of them may "stick" on with strange colors. Using `SetAllPixelsRGB()` with zeros is an easy way to make sure all lights are turned off, which clears any "stuck" pixels.

Like this:

```
SetAllPixelsRGB(0,0,0); //turns off all NeoPixel lights
```

A few more examples...

```
SetAllPixelsRGB(0,0,50); //all pixels to Blue  
SetAllPixelsRGB(50,0,0); //all pixels to Red
```

Use caution when using the `SetAllPixelsRGB()` function. As stated above, the pixels draw a bit of current, so trying to set them all to a high value at once will likely cause the over-current switch to trip. This isn't dangerous to Ringo, but it will make the lights go all wonky on you until Ringo is reset and loaded with new code that doesn't draw so much current.

Using Ringo's Motors



Ringo is fitted with a pair of small motors. These allow him to move around. The ends of the motors reach down and touch the surface he is sitting on. We can make him move forward, backward, or turn depending on how we control these motors.

Controlling the motors from your code is really easy. Let's jump right into an example then we'll talk about it.

```
void loop(){  
  SwitchSerialToMotors();    //Set the shared serial/motor line motors mode  
  Motors(LEFT, RIGHT);      //this function makes the motors go!  
}
```

Because the number of control lines available on Ringo's computer brain are limited, some of the control lines are shared with different functions. The lines that send serial data between Ringo and your computer are doubled up with the lines that control the direction of the motors. The line that says `SwitchSerialToMotors();` automatically configures these control lines to operate the motors. It's a good idea to include this line in your code anytime you are about to make the motors operate.

The second line is where the real action happens. The `Motors();` function accepts two numbers as arguments: one for left motor speed, and one for right motor speed. The values can range from -255 to positive 255 for each. When this line of code is executed, Ringo's motors will immediately start running at the commanded speed.

Let's see how this works with an example on the next page, but first, let's consider a few important notes.

Ringo is not an all-terrain robot, so he does require a smooth surface to move properly. Desktops, counter tops, and smooth flooring are ideal surfaces.

Be careful Ringo doesn't jump off your table. He does have sensors to see the edges of tables, but he won't respond to them unless your code specifically tells him to do so. We'll get to that eventually, but these simple examples do not automatically look for edges.

If Ringo's motors are forced to stop, they will not be damaged due to over-current, but it's still good practice to allow the motors to spin freely without forcing them to stop.

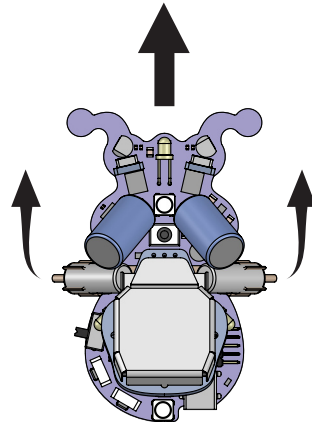
Using Ringo's Motors

Forward Movement

If we can make both motors turn in the forward direction, then Ringo should move forward. Let's try this example and see what happens:

```
void loop(){  
  SwitchSerialToMotors(); //Set the shared serial/motor line motors mode  
  Motors(200, 200);      // set left and right motors to 200 speed  
  delay(1000);            // wait 1 second  
  Motors(0, 0);          // make both motors stop  
  delay(3000);            // wait 3 seconds  
}
```

In this example, we use the command `Motors(200,200);` to set the left and right motors to a speed of 200. This makes Ringo start driving forward. Then we tell his brain to wait one second with `delay(1000);` so he will drive forward for one second. Then we set both of Ringo's motors to speed 0 with the command `Motors(0, 0);` which makes both motors stop moving. Finally, the command `delay(3000);` makes Ringo wait for three more seconds. After this final line of code, the loop will repeat again from the start.



Using Ringo's Motors

Backward Movement

Now let's make Ringo move backward. Backward movement works the same way as forward movement, you just give Ringo negative movement numbers. Try this example:

```
void loop(){  
  SwitchSerialToMotors(); //Set the shared serial/motor line motors mode  
  Motors(-200, -200);    // set left and right motors to NEGATIVE 200 speed  
  delay(1000);           // wait 1 second  
  Motors(0, 0);          // make both motors stop  
  delay(3000);           // wait 3 seconds  
}
```

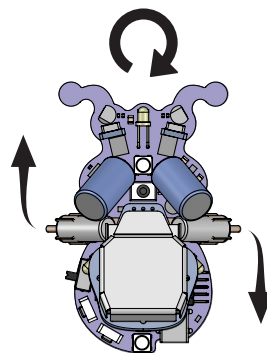
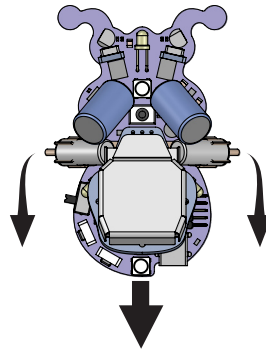
This example is exactly the same as the previous example, except that we've told Ringo to move the motors in the negative direction by placing minus signs in front of the speed numbers. The command `Motors(-200,-200);` does this for us.

Turning Movement

Have a guess at how we could make Ringo turn? We can set one motor to move forward and the other motor to move backward. Try using this command to set motor speed:

`Motors(200, -200);` What happens? This is causing the left motor to move at speed 200 in the forward direction, and the right motor to move at speed 200 in the backward direction (the minus sign makes the motor move backward).

What if we reverse these numbers? Try using this command to set motor speed: `Motors(-200, 200);` We now set the left motor to negative speed and the right to positive speed. This should cause Ringo to spin in the opposite direction.



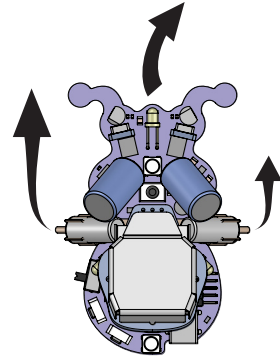
Using Ringo's Motors

Arc Movement

If you simply run both motors in opposite directions, we see Ringo turns around on his own footprint, which makes for a very agile little bug for sure. But what happens if you run both motors in the same direction, but at slightly different speeds?

Let's try running the motors like this:

`Motors(200, 100);` The left motor is now running about twice the speed of the right motor. Ringo still moves forward, but he is turning to the right.



Spend a few minutes playing with the numbers in the `Motors();` function. You'll notice you can change the arc depending on how you set the numbers.

"Straight" movement?

Lets go back to where we started in this discussion and look at forward movement a bit more closely. Let's try the example code below. Ringo is going to drive much further this time, so you may want to test this example on a floor. If you're doing this on a table, be ready to catch him before he jumps off the edge.

```
void loop(){
  SwitchSerialToMotors(); //Set the shared serial/motor line motors mode
  Motors(150, 150);      // make both motors run
  delay(3000);           // wait 3 seconds
  Motors(0, 0);          // make both motors stop
  delay(5000);           // wait 5 seconds
}
```

Did Ringo drive in a perfectly straight line? We set both motors to forward speed 150, so both motors should be turning at the same speed, right? You would think Ringo would have been driving in a perfectly straight line but I'm guessing he didn't (though you may have gotten lucky this time and perhaps he does travel perfectly straight, but not likely).

Using Ringo's Motors

Let Ringo drive this “straight” line a few times. You’ll probably notice he tends to veer off one direction or another. Study the Arc Movement example above and see if you can adjust the numbers you use in the `Motors();` function to make him go a bit more straight. Hint, if Ringo veers to the right, increase the right motor number a bit, and if he veers off the left, increase the left motor number a bit.

Mechanical Systems Aren't Perfect

Let's talk more about the previous experiment. Do you have any idea why Ringo didn't go straight even when the same motor speed numbers were used? The reason is that mechanical systems are never made perfectly. Each motor will run at a slightly different speed even when driven with the same power source as one will be slightly more efficient than the other. There are lots of other variables at play also - the surface may be more bumpy on one side, one motor tip may be getting better grip on the surface, etc. All mechanical systems have some variation. It's possible to minimize the variation (by using very expensive and super accurate motors for example), but some variation will always exist.

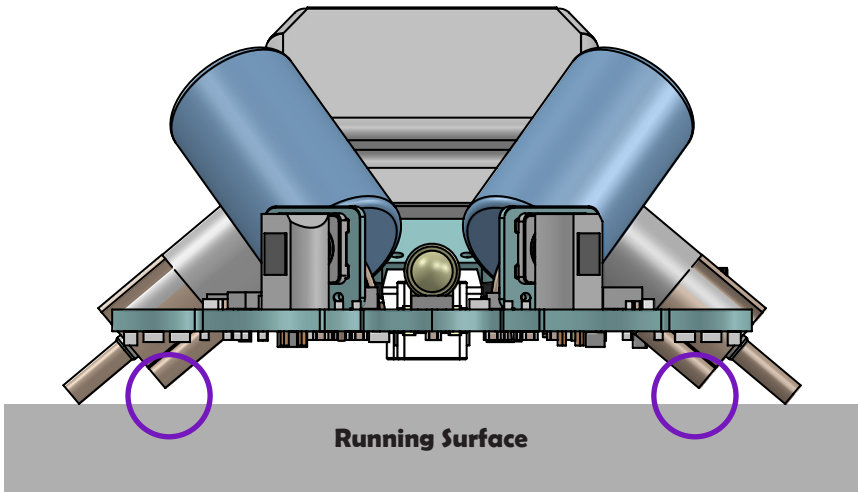
It isn't very practical to constantly guess and adjust a robot's speed settings to make it go in a straight line as you experienced in the last example. For this reason, most robots require some way of sensing how they are moving. Wouldn't it be smart if Ringo could somehow know that he was starting to veer to the right and automatically adjust the speed of his motors to compensate and resume a straight path? Well lucky for Ringo (and you!), he has two sensors that can be used for this purpose. One is called a Gyroscope, which basically allows Ringo to know when he is turning. The second one is called an Accelerometer, which basically allows Ringo to know how fast he is moving and how far he has traveled. Using these awesome sensors is beyond the scope of this lesson on basic motor operation, but check out the Navigation lesson later in this guide for the full run down.

Using Ringo's Motors

It no workey. (Sad Ringo).

The motors are the only real “mechanical” part of Ringo. It is important that they are oriented correctly to the running surface to operate properly. They may occasionally become misaligned in the motor clip, become bent, or get gummed up with hair or other yuk that Ringo may have driven through. Here is a quick guide to inspecting Ringo's motors and fixing any issues you may encounter.

1) Mis-Aligned Motor Clip: Make sure Ringo's motor clip is not dragging on the running surface. Look straight on to Ringo while he is on a surface. Look at the areas with the purple circles. There should be a gap of about 1mm between the running surface and the lowest portion of Ringo's motor clip.



If you find the motor clip is dragging, it can be corrected by making two adjustments.

- a) Make sure Ringo's motors are flush with the end of the motor clip. If the motors have been pushed further inward into the clip, the shaft is effectively shortened. Re-set motors to flush with the clip.
- b) The clip is spring steel and should be very difficult to deform, but you can carefully bend the ends of the motor clip downward. This causes a greater angle with the surface and thus creates more space between the clip and the surface.

Using Ringo's Motors

2) Bent Motor Shaft: This should not be a common problem, but it is possible. If Ringo takes a fall onto a hard surface and happens to land right on one of his motor shafts, it is possible the shaft could become bent. In all our testing, our Ringos have taken many hard falls onto a concrete shop floor from 3 to 4 feet heights. In all these drops, we managed to slightly bend one motor shaft which was easily fixed as follows. You'll notice it is bent because as Ringo moves, one side will "thump" along rather than running more smoothly. Correcting a bent motor shaft is possible with some care. Closely examine the shaft to determine the direction it is bent, then, using a tool of some sort, carefully reverse the bend. This will likely return Ringo to service. If the bend is so extreme that it cannot be reversed, motor replacement may be required. Contact us to order replacement motors.

3) Gunk on Motor Shaft: Ringo is very good at picking up debris in his motor shafts - especially hair from your pets when he is run on the floor. Usually this is easy to remove. Using tweezers, carefully pluck the hair off the shaft by rolling the shaft with your fingers. A bit of hair build up isn't a major problem, but if enough is allowed to accumulate, it can effect the performance of the bearing inside the motor where the shaft exits.

WARNING! If you do feel the need to remove a motor from the clip (this shouldn't be required, but we know some of you will try to do it anyway) - the best way to remove the motor is to slide it completely through the clip outward. It is tempting to rock the top end of the motor (where the wires come out) upward until it prys open the clip then rotate it outward. This is a bad idea though. As the motor is rotated out of the clip, all the force is placed on the motor shaft as the motor shaft hits the bottom of the clip. This force (if pryed all the way out of the clip) is enough to bend the motor shaft. Instead, slide it through the clip until it pops out the end. Be careful you don't break the wires when doing this. To re-install the motor, align it over the clip so the end is flush, then use finger pressure to press it into the clip from the top.

Chirps, Zips, and Pings! Sound!



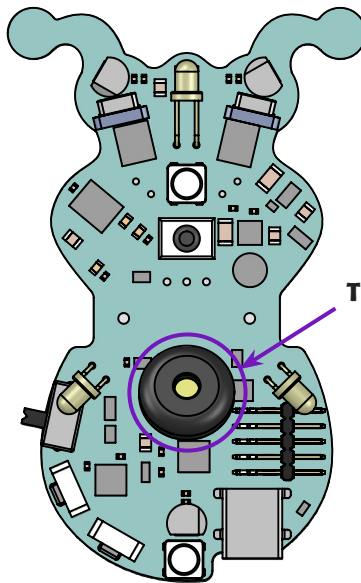
Ringo can make simple sounds with a piece of hardware called a piezo. The piezo is similar to speaker, and can be used to create tones.

Causing Ringo to emit a tone from his piezo is super easy. Just use the `PlayChirp()` function. The `PlayChirp` function takes two arguments. Arguments are bits of information you pass into a function when it is run. You did this with the `pixel` and `motor` functions in the previous example and didn't even realize it. :-)

Let's look at an example:

```
PlayChirp(Frequency, Amplitude);
```

You pass a frequency, which is the pitch of the tone to be played in hertz. Amplitude is a variable that controls the overall tonal quality and loudness. An amplitude of about 50 results in the loudest tone for most pitches. Normally you'll pass the amplitude as 50, but you can experiment with this number a bit to see what happens. Usually, amplitudes greater than 50 will result in more current being used (shorter battery life) without really making the piezo much louder.



This is Ringo's Piezo Element
(It is tucked under the battery)

1
SKILL
LEVEL

Chirps, Zips, and Pings! Sound!

Let's look at a working example. Try entering the following code into your main `loop()` function and see what happens.

```
void loop(){
  PlayChirp(2000, 50); //start playing a tone at 2000 hertz
  delay(1000);         //wait 1 second
  PlayChirp(4000, 50); //start playing a tone at 4000 hertz
  delay(1000);         //wait 1 second
  PlayChirp(0, 50);    //stop playing tone
  delay(2000);         //wait 2 seconds
}
```

Run the code on Ringo and see (hear?) what happens. You should hear a mid pitch tone for one second that then transitions to a higher pitch tone for one second, then you should hear silence for two seconds, then the process should repeat.

The line `PlayChirp(2000, 50);` causes the tone to begin. It should be noted that Ringo can do other things while a tone is playing. As you already know, the `delay(1000);` causes Ringo to wait for 1000 milliseconds (which is one second) before executing the next line `PlayChirp(4000, 50);` which causes Ringo to now begin playing the tone at 4000 hertz.

The `PlayChirp(0, 50);` is important because this is what is required to make Ringo stop playing a tone. You can make him stop playing a tone by either setting the frequency, or the amplitude, or both, to zero.

```
PlayChirp(0, 50); //set Frequency to zero. Ringo stops playing tone
PlayChirp(1000, 0); //set Amplitude to zero. Ringo stops playing tone
PlayChirp(0, 0); //set Frequency & Amplitude to zero. Ringo stops playing tone
```

Chirps, Zips, and Pings! Sound!

Playing Note Pitches:

If you want Ringo to play a song, you can tell him to play specific notes. Look at the tabs across the top of the Arduino IDE (or click the little down arrow at the end of the tabs if you can't see them all) and select the tab called "Pitches.h". This file "defines" each piano key note to the corresponding pitch in hertz. The "middle C" key of the keyboard is defined as "NOTE_C4". Let's make Ringo play middle C. Go back to the main tab in the Arduino IDE (the one on the far left) and try this code.

Like this:

```
void loop(){
  PlayChirp(NOTE_C4, 50);    //start playing the pitch for middle C
  delay(333);                //wait about 1/3 of a second
  PlayChirp(NOTE_E4, 50);    //start playing the pitch for E
  delay(333);                //wait about 1/3 of a second
  PlayChirp(NOTE_G4, 50);    //start playing the pitch of G
  delay(333);                //wait about 1/3 of a second
  PlayChirp(0, 0);           //stop making sound
  delay(2000);               //wait 2 seconds before looping
}
```

Ringo should be playing the C-Major chord every couple seconds. If you work with the notes and timings it is possible to make Ringo play simple songs.

Using Ringo's Light Sensors

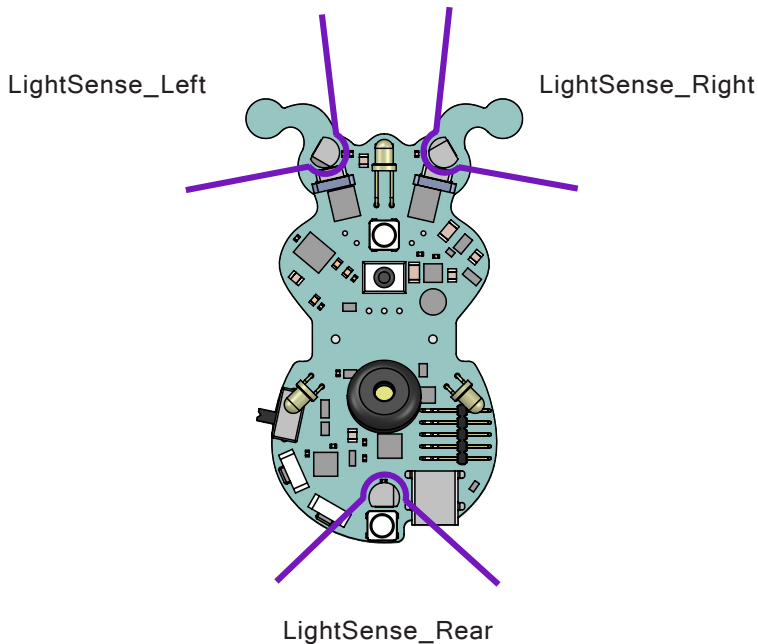


Ringo has some light sensors that he can use to observe his environment. These sensors can be used to control all kinds of behaviors and they are easy to use. Each sensor returns a value between 0 and 1024. The value goes up as more light reaches the sensor. Easy.

Setup...

Ringo's brain has three inputs that can read the analog light sensors. However, there are actually six light sensors on Ringo's body - there are the three ambient light sensors on the top side, and there are also three light sensors on the bottom side of Ringo to sense lines and edges. We can select which set of sensors are active by changing the `Source_Select` pin to either `HIGH` or `LOW`. Like this:

```
digitalWrite(Source_Select, HIGH); //This selects the top light sensors
digitalWrite(Source_Select, LOW);  //This selects the bottom light sensors
```



Using Ringo's Light Sensors

The best way to see how the sensors work is to use the Arduino `Serial.print` functions to send the light sensor readings to your computer screen so you can see how they change as you point lights at Ringo, or cast shadows on his sensors. Plug Ringo into his programming adaptor, then plug the programming adaptor into your computer. Go ahead and leave the adaptor connected to Ringo during this lesson. Let's load up the following code example and see what happens.

```
#include "RingoHardware.h" //include Ringo background functions

int sensorValue; //declares variable "sensorValue"

void setup(){
  HardwareBegin(); //initialize Ringo's circuitry
  PlayStartChirp(); //play startup chirp and blink eyes
}

void loop(){
  digitalWrite(Source_Select, HIGH); //select top side light sensors
  sensorValue = analogRead(LightSense_Right); //read right light sensor
  Serial.println(sensorValue); //print sensorValue through serial port
  Serial.println(); //print a blank line feed
  delay(250); //delay 250 milliseconds
}
```

In previous examples we simply told Ringo to do something. In this example, we're going to read a value from a sensor. Ringo needs to mark a location in his memory space to remember the value when it is read from the sensor. The line `int sensorValue;` "declares" the variable "sensorValue". Once the variable is declared, the variable can be used to store a value, to perform an operation on a value (like adding a number to it), or to recall the value from it.

We then use `digitalWrite` to set the `Source_Select` pin to `HIGH`, which enables the top side light sensors. Next we tell Ringo to perform an analog read of the `LightSense_Right` pin, and place the result of that analog read into our variable called `sensorValue`. An analog read simply creates a numerical value based on a voltage created by the light sensor. As the light is increased, this voltage is increased and thus, the result of the `analogRead` will increase.

Ringo now knows the present light level of his sensor (stored in the `sensorValue` variable), but you can't see it yet because it's only stored in Ringo's brain. The line `Serial.println(sensorValue);`

Using Ringo's Light Sensors

tells Ringo to send this value out his serial port to the programming adaptor, and the programming adaptor sends it to your computer. Open the serial monitor window and you will see the sensor value pop up in the window. The next line `Serial.println();` tells Ringo to print a line with nothing on it. This is basically a carriage return in your serial monitor window to create a space between readings. The last line `delay(250);` causes a quarter second delay between readings.

When you run this code, you should see a value between 0 and 1024 pop up in your serial monitor window four times per second. Let this run for a few seconds, then place your hand over Ringo's right eye. You should see the value go down a bit. It should go really low if you cover Ringo in a towel or blanket that will almost completely block out the light. If you turn him toward a bright light or window, you should see this value go up.

You may notice a few interesting things. First, it is impossible to bring this value all the way down to zero. This is beyond the scope of the lesson, but the light sensor always leaks a tiny bit of current which prevents it going all the way to zero. This is normal. In a similar way, the light sensors are designed to never totally max out the input of Ringo's brain, so even in direct sunlight you won't reach all the way to 1024.

You will also notice that the value goes up and down a bit even when the light in your room isn't changing. This is normal and can be caused by several things. Most room lighting actually flashes brighter and dimmer 60 times per second, which Ringo's sensors can pick up. This can also be caused by tiny bits of electronic "noise" coming from Ringo's power supply. If Ringo is charging his battery you'll see even more noise caused by the charging system.

A final item to note, is that Ringo's top side light sensors are "logarithmic". This basically means that they respond to differences in light levels the same way your eyes do, and they can see differences in a wide range of light ranging from very dim to very bright.

Using Ringo's Light Sensors

Three sensors at once...

Reading one sensor is great, but if we read more than one sensor, Ringo can make better decisions based on what he sees. Let's try an example where we read and serial print all three sensors at once. Can you guess how we would do this?

```
#include "RingoHardware.h" //include Ringo background functions

int sensorRight; //declares variable "sensorRight"
int sensorLeft;  //declares variable "sensorLeft"
int sensorRear;  //declares variable "sensorRear"

void setup(){
    HardwareBegin(); //initialize Ringo's circuitry
}

void loop(){
    digitalWrite(Source_Select, HIGH); //select top side light sensors
    sensorLeft = analogRead(LightSense_Left); //read left light sensor
    sensorRight = analogRead(LightSense_Right); //read right light sensor
    sensorRear = analogRead(LightSense_Rear); //read rear light sensor

    Serial.print(sensorLeft);           //print sensorValue through serial port
    Serial.print(" ");                  //print a couple blank spaces
    Serial.print(sensorRight);          //print sensorValue through serial port
    Serial.print(" ");                  //print a couple blank spaces
    Serial.print(sensorRear);           //print sensorValue through serial port
    Serial.println();                   //print a blank line feed
    Serial.println();                   //print a blank line feed
    delay(250);                         //delay 250 milliseconds
}
```

I think you can see what is happening here. We're using three variables now instead of just one. We read each sensor value into each variable, then use `Serial.print` to send it to your computer screen. Note in this example we're using `Serial.print()` rather than `Serial.println()`. The function `Serial.print()` basically prints information without a carriage return at the end, so all the values end up on the same line. The two `Serial.println();` functions at the end create a carriage return at the end of the output data, as well as a blank line between data sets.

Using Ringo's Light Sensors

Using top side light sensors when Ringo's eyes are lit up...

Have a close look at Ringo's eyes. You'll notice that a NeoPixel LED shines through the clear ambient light sensor. Whenever the NeoPixel is turned on (Ringo's eyes are lit up), this light from the NeoPixel is picked up by the top side light sensor directly in front of it. If you have Ringo's eyes lit up and want to read the light sensors, you will need to briefly turn off Ringo's eyes before reading the light sensors. The sensor read is very quick (about 100 microseconds per sensor) which is much too fast for your eye to see. If Ringo's NeoPixel eye lights are turned off, then the light sensors are read, then the eye lights are turned back on immediately following, it will appear that Ringo's eyes were lit up the entire time.

Try this code and see what happens:

```
#include "RingoHardware.h" //include Ringo background functions

int sensorRight; //declares variable "sensorRight"
int sensorLeft; //declares variable "sensorLeft"
int sensorRear; //declares variable "sensorRear"

void setup(){
  HardwareBegin(); //initialize Ringo's circuitry
  PlayStartChirp(); //play startup chirp and blink eyes
  digitalWrite(Source_Select, HIGH); //select top side light sensors
}

void loop(){
  SetAllPixelsRGB(0,0,0); //turn off all NeoPixel lights
  delay(2); //delay 2 milliseconds. Important!

  sensorLeft = analogRead(LightSense_Left); //read left light sensor
  sensorRight = analogRead(LightSense_Right); //read right light sensor
  sensorRear = analogRead(LightSense_Rear); //read rear light sensor

  // ... continued on next page ...
```

Using Ringo's Light Sensors

```
// ... continued from previous page ....

SetPixelRGB( 4, 220, 30, 160); //set pixel 4 (Right eye)
SetPixelRGB( 5, 220, 30, 160); //set pixel 5 (Left eye)
RefreshPixels();                //turn on the pixels

Serial.print(sensorLeft);       //print sensorValue through serial port
Serial.print(" ");              //print a couple blank spaces
Serial.print(sensorRight);      //print sensorValue through serial port
Serial.print(" ");              //print a couple blank spaces
Serial.print(sensorRear);       //print sensorValue through serial port
Serial.println();               //print a blank line feed
Serial.println();               //print a blank line feed
delay(500);                     //wait a half second before looping
}
```

Now we'll talk about what is happening. You'll notice a few differences in this example.

Firstly, we placed the `digitalWrite(Source_Select, HIGH);` in the `setup()` function because in this case, it only really needs to run once (as we're not switching to read the lower sensors in this case). This is fine to do, but you probably want to include this in your main `loop()` function if you're reading both top and bottom sensors.

We first start by turning off all the pixels. The next step `delay(2);` is super important. The light sensors are amplified through a part called an "op-amp", and this particular op-amp takes about a millisecond to adjust to the lower light level after Ringo's eyes are turned off. By delaying 2 milliseconds, we can be certain that Ringo is no longer seeing any "after glow" from having his eyes turned on. As soon as this delay is complete, we can go grab the three readings we want with `analogRead()` functions. As soon as the readings are complete, we flip Ringo's eyes back on. Once the eyes are back on, we send the serial data of the readings back to your computer, and finally, we wait a half second before doing it all again.

In the next section, we'll talk about reading Ringo's bottom sensors which can be used to sense lines and edges.

Using Ringo's Edge Sensors



Ringo has three light sensors on his under side. These sensors can be used to sense differences in the surface he is running on, such as edges of tables, lines, boxes, etc. These light sensors are intended to work together with three infrared light sources placed near the three light sensors.

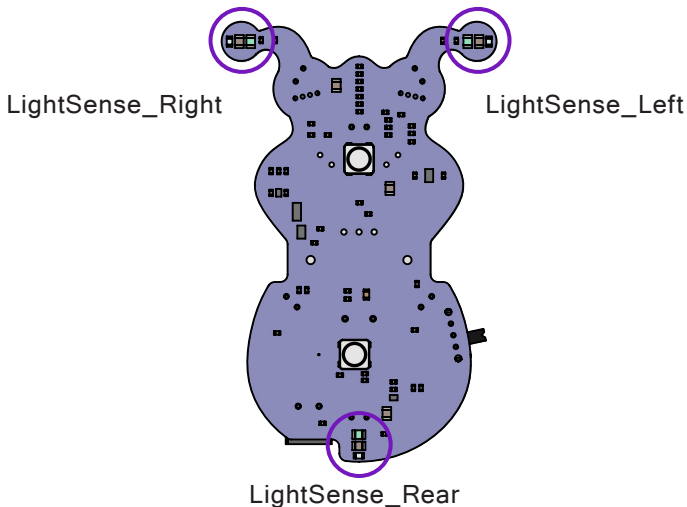
Setup...

As stated in the previous chapter, the LightSense lines to Ringo's brain are shared between the top "ambient" light sensors and the bottom light sensors. To select the lower light sensors, you need to set the `Source_Select` pin to `LOW`.

Like this:

```
digitalWrite(Source_Select, LOW); //This selects the bottom light sensors
```

RINGO BOTTOM SIDE

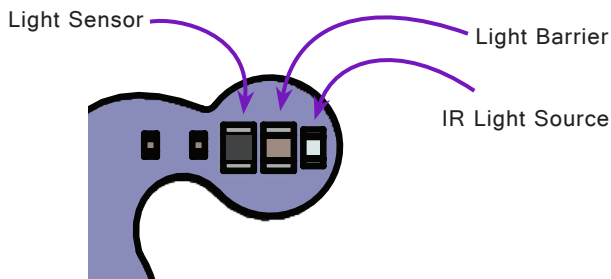


Using Ringo's Edge Sensors

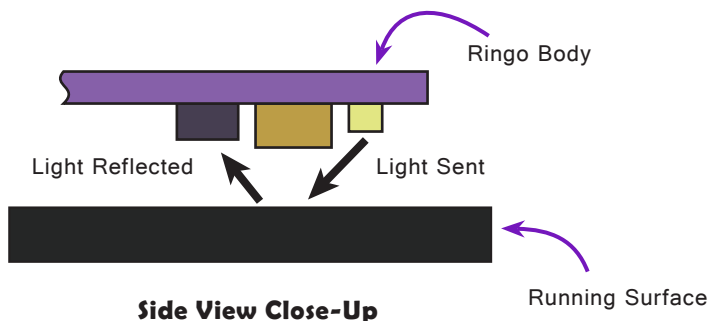
Next to each light sensor is a light source that produces infrared light. This light is invisible to your eye, but Ringo's edge sensor can see this light. All three lights are turned on and off together by setting the `Edge_Lights` pin. This infrared light is important. Lets discuss for a minute then it will be more clear what the code is actually doing. Don't let the details overwhelm you - later on in this section we present a couple super easy functions that automate the edge detection process.

How edges are measured...

The edge sensors are just light sensors mounted to Ringo's bottom side. If Ringo is placed on a white surface, these light sensors will see a bit more light, and if he is placed on a black surface, he will see less light. However, because Ringo's body casts a shadow on the surface, this measurement is only possible in well lit rooms. By adding extra light from the light sources on the `Edge_Lights` pin, we can do a neat trick to determine what sort of surface he is on.



Light Sensor Close-Up



Side View Close-Up

Using Ringo's Edge Sensors

We first turn on the infrared Edge_Lights sources. While these are turned on, we take a reading of one or more of the bottom light sensors and store the result in a variable. At this time, the light sensors are mostly seeing light reflected from the surface by the infrared light emitters. Each surface, material, or color will reflect a different amount of this infrared light.

We then turn off the edge light sources and take the same measurement again. This second measurement will show us the amount of ambient light being picked up by the edge light sensors (this is light from the room that may be getting into the sensor).

Once we have these two values, we can subtract them to determine the difference. This is essentially the amount of light reflected from the surface with the light from the room removed. Because each surface will reflect a different amount of light, this "difference" value can be evaluated and we can draw some assumptions about the surface from the "difference" value.

Darker surfaces will reflect less light, and brighter surfaces will reflect more light as you would expect.

We can now take this a step further and store a few historic "difference" readings. Each time we take a new reading, we can compare it to the last few readings. If the reading decreases suddenly, we can assume that the sensor has either moved over a darker feature in the surface (like a black stripe on the surface), or, that the sensor is hanging off the edge of the running surface - because there is no longer a surface present to reflect the light.

We have placed a capacitor between the light source and the light sensor. This capacitor is not electrically connected to anything. Instead we are using this capacitor as a "light barrier" which prevents the light from the light source from shining directly into the light sensor, so most of what the light sensor is seeing is in fact the light reflected from the surface.

Using Ringo's Edge Sensors

Using the bottom light sensors...

The bottom sensors can be read directly easily enough and used directly in your program. Consider this example:

```
#include "RingoHardware.h" //include Ringo background functions

void setup(){
    HardwareBegin();    //initialize Ringo's circuitry
    PlayStartChirp();    //play startup chirp and blink eyes
}

int leftOn, leftOff, rightOn, rightOff, rearOn, rearOff; //declare variables
int leftDiff, rightDiff, rearDiff;                      //more variables

void loop(){

    digitalWrite(Source_Select, LOW);                    // select bottom light sensors

    digitalWrite(Edge_Lights, HIGH);                    // turn on the IR light sources
    delayMicroseconds(200);                             // let sensors stabilize

    leftOn = analogRead(LightSense_Left);               // read sensors w/ IR lights on
    rightOn = analogRead(LightSense_Right);             // read sensors w/ IR lights on
    rearOn = analogRead(LightSense_Rear);               // read sensors w/ IR lights on

    digitalWrite(Edge_Lights, LOW);                    // turn off the IR light sources
    delayMicroseconds(200);                             // let sensors stabilize

    leftOff = analogRead(LightSense_Left);              // read sensors w/ IR lights off
    rightOff = analogRead(LightSense_Right);            // read sensors w/ IR lights off
    rearOff = analogRead(LightSense_Rear);              // read sensors w/ IR lights off

    leftDiff = leftOn-leftOff;                          // subtract out ambient light
    rightDiff = rightOn-rightOff;                       // subtract out ambient light
    rearDiff = rearOn-rearOff;                          // subtract out ambient light

    // ... continued next page ...                      // turn the page :-p
```

Using Ringo's Edge Sensors

```
// ... continued from previous page ...

Serial.print(leftDiff); // print the results to the serial window
Serial.print(" ");
Serial.print(rightDiff); // print the results to the serial window
Serial.print(" ");
Serial.print(rearDiff); // print the results to the serial window
Serial.println("");      // blank line with carriage return
delay(250);              // wait 1/4 second before doing it again
}
```

In this example we turn on the invisible infrared lights (sometimes called "IR" lights) to light up the surface. Then we wait 200 microseconds (2/10ths of a millisecond) to give the light sensors time to react and stabilize to the new lighting level. The `delayMicroseconds()` function is used for this. `delayMicroseconds()` is useful for delaying very short amounts of time. A microsecond is a millionth of a second. It is good practice to add a slight delay between changing the input to a sensor (like the light level) and reading the sensor as it may take a short time for the sensor to stabilize to the new input level. After we let the sensors stabilize to the new light level, we read each of them.

We then do the same process after turning off the IR lights. Finally we subtract out the ambient light level and send the results to your serial monitor window so you can see the readings.

Experiment with Ringo over different surfaces and see how the number changes. The first number in the serial window is the front left, then front right, then rear sensor value. As you place a sensor over a different material you will see this value change. Hold him a ways off the surface and these numbers will likely go under 100. On a black surface or dark desk, you'll probably see 100 to 150, and if you place him on a white piece of paper, the values should shoot up to 800 or more.

It is easy to make Ringo react in different ways to these values. For example, if you run him on a white surface and surround it with black tape, you could turn on his motors and repeatedly evaluate if the reading remains over 500, and if it ever drops below 500, you stop the motors. This would cause Ringo to drive until he reached the tape line.

Using Ringo's Edge Sensors

Edge Sensing Functions (the easy way)...

The previous example would be great if you knew that one surface will produce one value and the “different” surface would produce a different value and you were certain what those values would be. However, in the real world, Ringo may find himself on any variety of surfaces. What is really useful is to look for a sudden change in sensor readings. Lets say he goes from grey to white which may cause the value to go from 600 to 675. Or if he's on a dark color table and begins to run off the edge, which may cause the value to go from 120 to 85. If we can store a historic average of readings, then upon each new reading we compare this new reading to the running historic average, we can easily detect sudden changes.

We've written a few functions that do this for you.

```
ResetLookAtEdge(); // run this once before using the other edge functions
LookForEdge();     // reads all edge sensors, automatically compares
                   // to running averages, returns which sensors
                   // have seen edges
```

Ringo keeps a running average of each edge sensor reading. By calling `ResetLookAtEdge()` this running average is reset. You can then call `LookForEdge()` as often as you like. The `LookForEdge()` function calls a second function called `LookAtEdge()` which actually takes the readings. Once the readings are measured, `LookForEdge()` determines if any sudden changes have been observed. `LookForEdge()` returns “0” if no edge has been detected. If an edge has been detected, it returns a value where each bit represents which edge sensor saw the detection, and whether that detection was a bright or dark edge.

The bits are mapped as follows:

```
0: right dark edge (0x01)   1: left dark edge (0x02)
2: rear dark edge (0x04)    3: right bright edge (0x08)
4: left bright edge (0x10)  5: rear bright edge (0x20)
```

The Arduino bitwise operators can be used to evaluate the result of `LookForEdge()` to determine which sensor was triggered. For more information on bitwise operators, visit: <https://www.arduino.cc/en/Tutorial/BitMask>

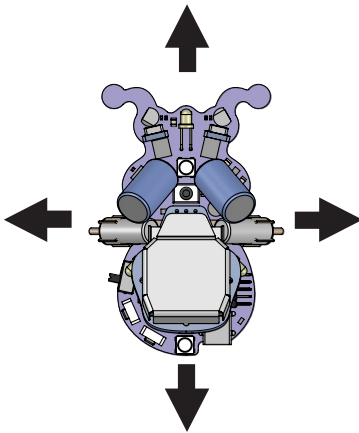
Using Ringo's Navigation



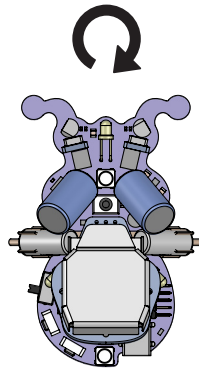
Ringo has two incredible sensors that can be used to sense his motion. The first sensor is called an Accelerometer. An Accelerometer senses movement in front-back, left-right, and up-down motion. It can be used to determine how far Ringo moves. The second sensor is called a Gyroscope. A Gyroscope senses rotational movement. For example, if you keep Ringo in the same location but you rotate him in a turn, the Gyroscope can be used to determine how many degrees he has rotated.

Taken together, these sensors embody an “inertial navigation” system, where Ringo can determine how far he has moved and how his orientation is changing. You could for example, make a 45 degree turn and move forward 1 meter. Awesome!

This is the first “Skill Level 2” lesson. We’ll get a bit more advanced in the discussion here but have no fear - the actual functions used to move Ringo are fairly straightforward.



Accelerometer



Gyroscope

2
SKILL
LEVEL

Using Ringo's Navigation

Background Information...

Let's start with some background information about how these sensors work. Don't let this part of the discussion scare you - this is good information to help you understand how and why the movement functions work the way they do.

The accelerometer doesn't actually tell you where Ringo is or how far he has moved. The sensor simply measures acceleration - which is how quickly Ringo's speed is changing at any given instant. It measures this 800 times per second, and stores the measurements in a memory inside the sensor. Every so often (about every 80 milliseconds in Ringo's case), Ringo's brain needs to establish a data connection with the sensor and unload these values. Because we know how much time elapsed between each measurement, Ringo can do some math and determine that a certain acceleration force will result in a certain speed, and that taken over time can determine distance. If the sensor data isn't read out every 80 milliseconds, the memory inside the sensor (called a "buffer") will "overflow". The overflow doesn't hurt anything, but it will forget the values it has already taken which means when Ringo eventually runs the math on those samples, the end result may not be accurate.

The gyroscope works in a similar way. It doesn't actually tell you what direction Ringo is facing. Instead, it takes snapshots of "angular velocity" at regular intervals timed very close together and stores those values in its own buffer. Ringo establishes a data connection to the gyroscope and unloads these values. Again, some math can be done on these values and determine that a given angular velocity over a given period of time will result in a predictable number of degrees of rotation and thus, Ringo can keep up on what direction he is facing.

Lucky for you, this process of reading values out of the sensors and performing the math on them (which is rather complex) has already been worked out and is performed automatically by calling a few simple functions.

Using Ringo's Edge Sensors

Here is a quick list of the navigation functions along with a short description of each. These functions take care of the data communication between Ringo's brain and the sensors as well as performing the required math on the resulting data so they are easy to use.

Base setup functions:

NavigationBegin(); This function turns on and initializes the gyroscope and accelerometer. It is important to call this function before running the other navigation functions. It can be called over and over with no harmful effects.

CalibrateNavigationSensors(); This function calibrates the sensors. Even when the sensors are perfectly still, they do have a slight offset because no sensor is perfect. To calibrate the sensors, we measure some samples when they are still to see what this offset is. Ringo then remembers this offset and subtracts it out of future measurements. The NavigationBegin() function calls this CalibrateNavigationSensors(), so you don't need to call it a second time. It is important that Ringo is perfectly still when calling this CalibrateNavigationSensors(). Vibrations or wiggles on your table will adversely effect the calibration. The sensors should be recalibrated every so often. Keep in mind that it is useful to place a delay() before each calibration to make sure Ringo isn't still moving from a previous motor operation, etc. Wait a second or so to calibrate after the Reset or User buttons are pressed as the presense of your finger on the robot will adversely effect the calibration accuracy.

ZeroNavigation(); Ringo keeps a memory of his X and Y location as well as his rotation. The X location is left/right of his starting location, with a positive number to the right, and a negative number to the left. The Y location is front/rear of his starting location with positive numbers in the forward direction and negative numbers in the rear direction. Units are in millimeters of travel. Rotation is in units of degrees. Positive degrees are in the right (clockwise) direction and negative degrees are in left (counter-clockwise) direction. Note that the degrees do not loop. For example, if you rotate one and a half turns in the clockwise direction, the resulting rotation value will be 540. The ZeroNavigation() function zeros these running X, Y, and rotation numbers to zero.

GetHeading(); The GetHeading() function returns Ringo's present heading in degrees.

Using Ringo's Edge Sensors

Movement Functions:

Ringo's inertial navigation can be used in lots of interesting ways, and we encourage you to explore the possibilities. The most obviously useful function is in controlling Ringo's motors to move him in predictable ways. We have written a selection of functions that move Ringo's motors while reading the navigation sensors to go certain distances, turn in certain ways, etc. These functions handle running the motors including speed and direction, reading the sensors, performing the math on the sensor data, and adjusting motor speed to achieve the desired movement.

All movement functions use a "PID" software loop, which is a software technique commonly used to move robot arms. Basically, a PID loop causes the motors to move more quickly if they are further from their target, and more slowly as they reach the target. For example, when rotating, if Ringo needs to turn 180 degrees, the motors will run quickly at first and will slow progressively as Ringo nears the 180 degree point. So the speed is naturally self correcting and minimizes overshoot.

Rotation:

RotateSimple() This function causes Ringo to perform a turning movement or rotation. It is not quite as accurate as the next rotation example but will arrive at the target rotation a bit more quickly. The function takes five arguments and is prototyped as follows.

```
void RotateSimple(int TurnDegrees, int left, int right, \
                  int MaxExpectedTurnTime, int MaxExpectedSkidTime);
```

The terms work as follows. **TurnDegrees** is the number of degrees to be rotated during the turn. Positive or negative numbers can be used. **left** and **right** are the speed for each motor. This controls how fast Ringo will rotate. If **left** and **right** are equal numbers, Ringo will rotate in his own footprint. If **left** and **right** are not equal, Ringo will perform an arc drive as he moves toward the desired rotation target. **MaxExpectedTurnTime** is the time (in milliseconds) Ringo will allow to reach his target rotation before giving up and moving on in the code. Without this feature Ringo would drive his motors as long as required until his sensors tell him that he has arrived at the target rotation. If you tell him to rotate 180 degrees, but his motors get hung up in a crack, he would sit there forever trying to reach 180. Normally you'll want to make this value slightly longer than you expect the maneuver will take to complete. Once Ringo arrives at the target rotation, he will turn off his motors. Because there is

Using Ringo's Edge Sensors

some rotational inertia to Ringo and his motors, he won't stop immediately. He will skid on the surface for some period of time. Once the motors are turned off, Ringo will continue to evaluate his gyroscope and wait for the readings to indicate that he has in fact come to a complete stop. Similar to the `MaxExpectedTurnTime`, the `MaxExpectedSkidTime` is the maximum time Ringo will remain looking for a complete stop. This time can be fairly short as he will normally stop very quickly and move ahead to the next instruction on his own.

Lets consider a working example:

```
#include "RingoHardware.h" //include Ringo background functions

void setup(){
    HardwareBegin(); //initialize Ringo's circuitry
    PlayStartChirp(); //play startup chirp and blink eyes
    delay(1000); //wait 1 second after restart before starting navigation
    NavigationBegin(); //initialize navigation system
}

void loop(){
    //rotation, left, right, turntime, skidtime
    RotateSimple(180, 200, 200, 1500, 100); // rotate clockwise @ speed 200
    delay(1000); // wait 1 second
    //rotation, left, right, turntime, skidtime
    RotateSimple(-180, 200, 200, 1500, 100); // rotate counter-clock @ speed 200
    delay(1000); // wait 1 second
}
```

In this example, we call `NavigationBegin()` one time in the setup function. In the main loop, we first tell Ringo to rotate 180 degrees clockwise, at motor speed 200. He will attempt to reach 180 degrees for 1.5 seconds. If he cannot reach 180 in this time, he will skip out of the `RotateSimple()` function and proceed to the next line of code. Once he stops, he will wait 1/10th of a second to reach a complete stop. These turn and skid times should be plenty to reach the target. The loop then makes the same movement in reverse and continues in a loop.

Hint: When you experiment on your own, if you see Ringo consistently fails to turn far enough, have a look at the `MaxExpectedTurnTime` argument and make sure you're allowing him enough time to complete the maneuver.

2
SKILL
LEVEL

Using Ringo's Edge Sensors

RotateAccurate() This function is similar to the **RotateSimple()** function, except that it moves a bit more slowly and if Ringo over-shoots the target rotation, he will reverse direction and attempt to reach the target rotation more accurately. It is prototyped as follows.

```
char RotateAccurate(int Heading, int MaxExpectedTurnTime);
```

The **RotateAccurate()** function accepts two arguments: **Heading** and **MaxExpectedTurnTime**. **Heading** is the target rotation, and like the last example, the **MaxExpectedTurnTime** is the max time Ringo will attempt to complete the maneuver.

Movement:

MoveWithOptions() This is the most basic function for movement. It accepts seven arguments and is prototyped as follows.

```
void MoveWithOptions(int Direction, int Speed, int Distance, \
    int MaxExpectedRunTime, int MaxExpectedSkidTime, \
    void (*EdgeFunction)(char), char Wiggle);
```

The **MoveWithOptions()** function looks a bit intimidating, but it's actually pretty straightforward. Let's discuss it. The first argument is **Direction**. Ringo will attempt to turn to this direction as he begins driving. **Speed** is the average motor speed he should try to maintain. This will be adjusted up or down slightly between the two motors to produce the desired direction. **Distance** is the overall forward or backward travel distance in millimeters. As with the previous functions, **MaxExpectedRunTime** is the maximum time Ringo will allow to carry out the attempted maneuver, and **MaxExpectedSkidTime** is the maximum time Ringo will wait to arrive at a complete stop once he arrives at the commanded destination. The **(*EdgeFunction)(char)** term allows Ringo to call a certain function if he reaches an edge during his travel. That's a bit more advanced and we'll cover it later, but for now, just pass a **0** in this argument. The **Wiggle** argument controls how much "buggy swagger" Ringo will show. A **0** in this argument will cause him to move fairly straight with only a minimal wiggle. A **50** in this argument looks pretty buggy - he sways around a lot more. It is suggested that a maximum value of **100** be passed for this argument. Anything more than **100** will cause him to have a real problem maintaining course, but it can be interesting to watch so go ahead and experiment with it.

Lets have a look at a working example of this function.

```
#include "RingoHardware.h" //include Ringo background functions

void setup(){
    HardwareBegin();    //initialize Ringo's circuitry
    PlayStartChirp();    //play startup chirp and blink eyes
    delay(1000);         //wait 1 second after restart before starting navigation
    NavigationBegin();   //initialize navigation system
}

void loop(){
    //direction, speed, distance, runtime, skidtime, edge, wiggle
    MoveWithOptions(0, 220, 300, 2000, 100, 0, 10); // go forward 300 mm
    delay(1000);                                     // wait 1 second
    MoveWithOptions(0, -220, -300, 2000, 100, 0, 10); // go backward 300 mm
    delay(1000);                                     // wait 1 second
}
```

In this example, we call `MoveWithOptions()` and tell it to drive Ringo in the `0` degree heading (straight ahead) at a motor speed of `220` (motor speeds can be positive or negative `0` to `255`) for a distance of about 300 mm. He will allow 2 seconds to complete the maneuver, allow 100 milliseconds to come to a complete stop after the maneuver, and he'll do it all with level 10 buggy swagger. He then does the same in reverse. To drive in reverse you'll specify a negative motor speed (`-200` in this case) as well as a negative distance (`-300` in this case).

Now lets combine movement and rotation. Like this:

```
void loop(){
    //direction, speed, distance, runtime, skidtime, edge, wiggle
    MoveWithOptions(GetHeading(), 220, 300, 2000, 100, 0, 10);
    // go forward 300 mm at present heading
    delay(500);                                     // wait 1/2 second
    RotateAccurate(90, 2000);                       // rotate 90 degrees clockwise
    delay(500);                                     // wait 1/2 second
}
```

In this example, we've made a change to the `MoveWithOptions()` function. You may notice that instead of going direction `0`, we instead have plugged another function into this spot - `GetHeading()`. Lets talk about this for second.

Lets say we put a `0` instead of `GetHeading()`. You may want to try this and see what happens on your own. When we put a `0` in this function, Ringo is going to try and drive to the heading of `0` referenced from the last time you called either `HardwareBegin()` or `ZeroNavigation()`. Putting a `0` in here the first time through the loop causes him to drive straight as expected. But after we rotate him 90 degrees, he is no longer headed in the `0` degree direction. If we call `MoveWithOptions()` a second time after the rotation, and we put a `0` for direction, Ringo will make a left turn as he starts to drive because he is trying to return to the `0` degree direction.

An easy way to avoid this, is to always place `GetHeading()` in place of the zero. The `GetHeading()` function reads Ringo's current heading and automatically passes this current heading to the `MoveWithOptions()` function each time it is called. By passing `GetHeading()` for the direction, Ringo will always travel in a straight line each time the function is called, no matter how much you rotate him.

You could do other cool things like passing "`GetHeading()+15`" which would cause Ringo to make a 15 degree clockwise turn during each movement.

These are the basics to get you started. We will continue to fill out this guide with additional examples and greater explanation as we move forward. Cheers!!