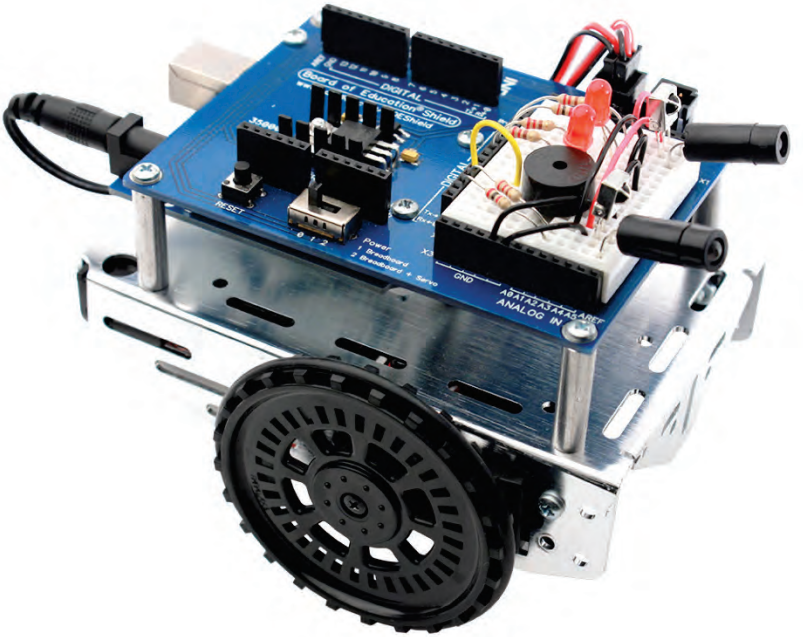
Robotics with the BOE Shield-Bot for Arduino



by Andy Lindsay

Version 1.1

WARRANTY

Parallax warrants its products against defects in materials and workmanship for a period of 90 days from receipt of product. If you discover a defect, Parallax will, at its option, repair or replace the merchandise, or refund the purchase price. Before returning the product to Parallax, call for a Return Merchandise Authorization (RMA) number. Write the RMA number on the outside of the box used to return the merchandise to Parallax. Please enclose the following along with the returned merchandise: your name, telephone number, shipping address, and a description of the problem. Parallax will return your product or its replacement using the same shipping method used to ship the product to Parallax.

14-DAY MONEY BACK GUARANTEE

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a full refund. Parallax will refund the purchase price of the product, excluding shipping/handling costs. This guarantee is void if the product has been altered or damaged. See the Warranty section above for instructions on returning a product to Parallax.

COPYRIGHTS AND TRADEMARKS

Copyright © 2012-2016 by Parallax Inc. All Rights Reserved. By obtaining a printed copy of this documentation or software you agree that it is to be used exclusively with Parallax Board of Education Shield for Arduino products. Any other uses are not permitted and may represent a violation of Parallax copyrights, legally punishable according to Federal and international copyright or intellectual property laws. Any duplication of this documentation for commercial uses is expressly prohibited by Parallax Inc. Duplication for educational use, in whole or in part, is permitted subject to the following conditions: the material is to be used solely in conjunction with Parallax microcontrollers and products, and the user may recover from the student only the cost of duplication. Check with Parallax for approval prior to duplicating any of our documentation in part or whole for any other use.

BASIC Stamp, Board of Education, Stamps in Class, and Boe-Bot are registered trademarks of Parallax Inc. BOE Shield-Bot, HomeWork Board, PING))), Parallax, the Parallax logo, Propeller, and Spin are trademarks of Parallax Inc. Arduino is a registered trademark of Arduino, LLC. Other brand and product names herein are trademarks or registered trademarks of their respective holders.

ISBN 978-1-928-98261-6

1.1.0-161102-FL

DISCLAIMER OF LIABILITY

Parallax Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, or any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax is also not responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your microcontroller application, no matter how life-threatening it may be.

ERRATA

While great effort is made to assure the accuracy of our texts, errors may still exist. Occasionally an errata sheet with a list of known errors and corrections for a given text will be posted on the related product page at www.parallax.com. If you find an error, please send an email to editor@parallax.com.

Table of Contents

About This Tutorial 6
About This Edition 6
Why Study Robotics? 6
A Bit of Background 7
Audience & Support 8
Author & Contributors 8
Chapter 1. Your Shield-Bot's Brain 9
Robot Kit and Software Options 9
Check Your Hardware 11
Activity 1: Download Example Sketches and Software 16
Activity 2: Write a Simple "Hello!" Sketch 17
Activity 3: Store and Retrieve Values 24
Activity 4: Solve Math Problems 27
Activity 5: Make Decisions 30
Activity 6: Count and Control Repetitions 32
Activity 7: Constants and Comments 36
Chapter 1 Summary 37
Chapter 1 Challenges 38
Chapter 2. Shield, Lights, Servo Motors 42
Activity 1: Board of Education Shield Setup 42
Activity 2: Build and Test LED Indicator Lights 47
Activity 3: LED Servo Signal Monitors 55
Activity 4: Connect Servo Motors and Batteries 60
Activity 5: Centering the Servos 63
Activity 6: Testing the Servos 65
Chapter 2 Summary 71
Chapter 2 Challenges 72
Chapter 3. Assemble and Test your BOE Shield-Bot 76
Activity 1: Assembling the BOE Shield-Bot 76

Activity 2: Re-Test the Servos 85
Activity 3: Start-Reset Indicator 89
Activity 4: Test Speed Control 93
Chapter 3 Summary 99
Chapter 3 Challenges 100
Chapter 4. BOE Shield-Bot Navigation 103
Activity 1: Basic BOE Shield-Bot Maneuvers 104
Activity 2: Tuning the Basic Maneuvers 109
Activity 3: Calculating Distances 112
Activity 4: Ramping Maneuvers 116
Activity 5: Simplify Navigation with Functions 118
Activity 6: Custom Maneuver Function 125
Activity 7: Maneuver Sequences with Arrays 127
Chapter 4 Summary 139
Chapter 4 Challenges 140
Chapter 5. Tactile Navigation with Whiskers 147
Activity 1: Build and Test the Whiskers 148
Activity 2: Field-Test the Whiskers 154
Activity 3: Navigation with Whiskers 157
Activity 4: Artificial Intelligence for Escaping Corners 162
Chapter 5 Summary 168
Chapter 5 Challenges 168
Chapter 6. Light-Sensitive Navigation with Phototransistors 176
Introducing the Phototransistor 177
Activity 1: Simple Light to Voltage Sensor 178
Activity 2: Measure Light Levels Over a Larger Range 188
Activity 3: Light Measurements for Roaming 197
Activity 4: Test a Light-Roaming Routine 202
Activity 5: Shield-Bot Navigating by Light 207

Chapter 6 Summary 209
Chapter 6 Challenges 210
Chapter 7. Navigating with Infrared Headlights 216
Infrared Light Signals 216
Activity 1: Build and Test the Object Detectors 218
Activity 2: Field Testing 224
Activity 3: Detection Range Adjustments 230
Activity 4: Object Detection and Avoidance 232
Activity 5: High-performance IR Navigation 235
Activity 6: Drop-off Detector 238
Chapter 7 Summary 243
Chapter 7 Challenges 244
Chapter 8. Robot Control with Distance Detection 249
Determining Distance with the IR LED/Receiver Circuit 249
Activity 1: Testing the Frequency Sweep 250
Activity 2: BOE Shield-Bot Shadow Vehicle 254
Activity 3: What's Next? 262
Chapter 8 Summary 265
Chapter 8 Challenges 265
Index 268

About This Tutorial

New to robotics? No problem! The activities and projects in this text start with an introduction to the BOE Shield-Bot's brain, the Arduino® Uno. Next, you will build, test, and calibrate the BOE Shield-Bot. Then, you will learn to program the BOE Shield-Bot for basic maneuvers. After that, you'll be ready to add different kinds of sensors, and write sketches to make the BOE Shield-Bot sense its environment and respond on its own.

New to microcontroller programming? This is a good place to start! The code examples introduce Arduino programming concepts little by little, with each example explained fully.

New to electronics? Each electronic component is described with a circuit symbol and part drawing. Traditional schematics next to wiring diagrams make it easy to build the circuits.

About This Edition

Version 1.1 was updated for use with codebender : edu, as was the web tutorial posted at http://learn.parallax.com/shieldrobot as of 11/02/2016. Editorial changes to accommodate the print format were made, but the educational content remains closely parallel to the web tutorial. Also, this edition was narrowed by ½ inch to reduce paper waste in the manufacturing process.

Why Study Robotics?

Robots have been in use for all kinds of manufacturing and in all manner of exploration vehicles—and in many science fiction films—for a long time. The word 'robot' first appeared in a Czechoslovakian satirical play, Rossum's Universal Robots, by Karel Capek back in 1920! Robots in this play tended to be human-like, and much science fiction that followed involved these robots trying to fit into society and make sense out of human emotions.

Then, General Motors installed the first robots in its manufacturing plant in 1961. Those automated single-purpose machines presented an entirely different image from the human-like robots of science fiction. As technology continues to advance, increasingly human-like robots designed for socially oriented tasks are emerging, and many types of robots co-exist today.

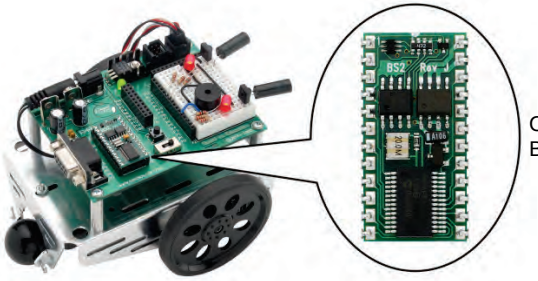
Regardless of a robot's outer form, building and programming most robots requires a combination of mechanics, electronics, and problem-solving. What you can learn from this tutorial will be relevant to real-world robot applications. Of course there will be differences in size and sophistication, but the underlying mechanical principles, basic circuits, and programming concepts are used by engineers every day. New uses for robots are emerging constantly. Roboticists are in demand!

A Bit of Background

The BOE Shield-Bot is a small microcontroller development platform on wheels. It is designed for experimenting with circuit-building and programming, using standard electronic components.

Board of Education Shield-Bot with Arduino Uno brain

It is a variation of the original Boe-Bot® robot with its BASIC Stamp® 2 brain, shown below. It was introduced by Parallax Inc. in 1999 and continues to be widely used with schools for STEM courses, because PBASIC is a very easy language for a first-time, text-based programming experience.



Original Boe-Bot Robot with BASIC Stamp 2 brain

The Arduino microcontroller arrived on the scene in 2005 and its popularity grew through the DIY (do-it-yourself) hobby community. Parallax teamed up with SimplyTronics to design the Board of Education® Shield, which makes the Arduino hardware compatible with the Boe-Bot chassis. The Arduino can do the same tasks as the BASIC Stamp, though in a slightly different way, but it was still straightforward to rewrite the original *Robotics with the Boe-Bot* to support the Shield-Bot. This gives the Arduino hobby community the opportunity to wrap a robotics shield around an Uno. It also gives teachers a different programming language option for working with Parallax robots.

Audience & Support

This book is designed to promote technology literacy through an easy introduction to microcontroller programming and simple robotics.

Are you a middle-school student? You can be successful by following the check-marked instructions with your teacher's support.

If you are a pre-engineering student, push yourself a little farther and test your comprehension and problem-solving skills with the questions, exercises, and projects in each chapter summary (solutions are provided!)

If you are an independent learner working on your own, go at your own pace and check in with Parallax's Robotics forum (forums.parallax.com) if you get stuck.

If you are an educator, feel free to contact our team at education@parallax.com for additional resources and support deploying the Shield-Bot in your STEM program.

Author & Contributors

About the Author

Andy Lindsay joined Parallax Inc. in 1999, and has since authored more than a dozen books, including *What's a Microcontroller?* as well as numerous articles, product documents, and web tutorials for the company. The original *Robotics with the Boe-Bot* that is the inspiration for this book was designed and updated based on observations and educator feedback that Andy collected while traveling the nation and abroad teaching Parallax Educator Courses and events. Andy studied Electrical and Electronic Engineering at California State University, Sacramento, and is a contributing author to several papers that address the topic of microcontrollers in pre-engineering curricula. When he's not writing educational material, Andy does product and applications engineering for Parallax.

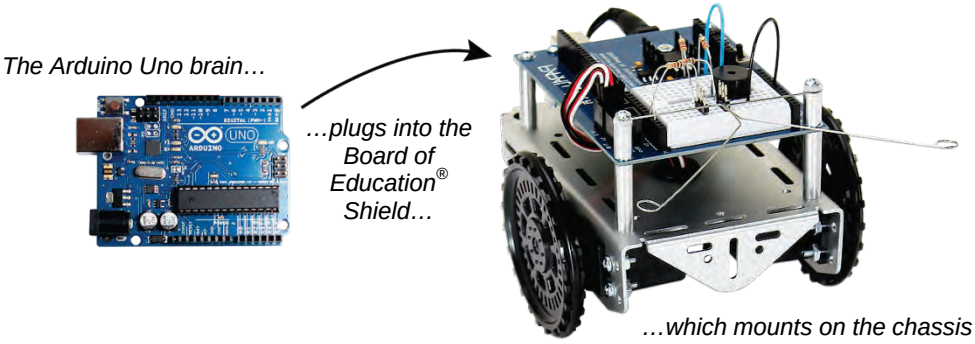
Special Contributors

The Parallax team assembled to prepare this edition includes technical writing by Andy Lindsay, cover art by Jen Jacobs, graphic illustrations by Courtney Jacobs and Andy Lindsay, photography by Robert Reimers, editing and layout by Stephanie Lindsay.

Several customers helped test-drive this material. Thanks go to Gordon McComb for test-driving and technical feedback on the original chapter drafts. Special thanks also go to Matt Zawlocki and his Fall 2015 middle-school students for trying all of the example sketches with the original Codebender browser-based program editor.

Chapter 1. Your Shield-Bot's Brain

Parallax Inc.'s BOE Shield-Bot robot is the focus of the activities and projects in this book. The robot's Board of Education Shield mounts on a metal chassis along with servo motors, wheels, and a battery pack. An Arduino Uno module—the programmable brain—plugs in underneath the BOE Shield.



The activities in this tutorial will guide you through building the mechanical parts and circuits that make the BOE Shield-Bot work. Then, you'll write simple programs that make the Arduino and your robot do four essential robotic tasks:

- Monitor sensors to detect the world around it
- Make decisions based on what it senses
- Control its motion (by operating the motors that make its wheels turn)
- Exchange information with its roboticist (that will be you!)

Robot Kit and Software Options

To do the activities in this tutorial, you will need both hardware and software.

Robot Shield Kit Options

- Robot Shield Kit with Arduino (includes Arduino Uno and USB A-B cable; #32335)
- Robotics Shield Kit for Arduino (no Arduino nor USB A-B cable; #130-35000)

Boe-Bot Retrofit: If you have a BASIC Stamp Boe-Bot robot, you can convert it to a Shield-Bot with a Retrofit Kit, available with an Arduino Uno (#910-32333) or without (#32333); USB programming cable sold separately (#805-00007).

Arduino Module Options

The Arduino Uno is the preferred module for the Shield-Bot robot.

This tutorial was also tested with a Duemilanove and original Mega. These Arduino modules automatically decide whether to draw power from USB or an external source (like the Shield-Bot's battery pack).

Older Arduino Models are not guaranteed to work. If you have one, you may have to set its power selection jumper. (Don't worry about this if you have an Uno, Duemilanove, or Mega.) The circuit is labeled PWR_SEL. It is three pins with a small cover called a shunt that slides over two of three pins. For now, make the shunt cover the USB and center pins. Later, when you switch to using the Shield-Bot's battery pack, move the shunt to cover the EXT pin and center pin instead.

Software Options

This tutorial requires the Arduino language 1.0 or higher. There are two recommended software options for using this language.

- **Arduino IDE** is a development environment software package that gets installed on your computer, and you do not have to be online to use it.
- **Codebender : edu** is a very easy to use, web-browser-based editor for Arduino sketches. It offers classroom subscriptions, and it is accessible online from a variety of operating systems and browsers.

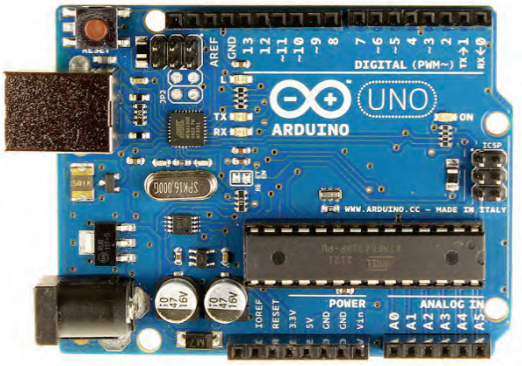
If this is your first time using an Arduino, Activity #1 will help you get started with your choice of software, connecting your hardware, and testing your programming connection. The rest of this chapter includes a series of example programs (called *sketches*) that introduce common programming concepts. The sketches will do some of the most basic yet important things for a robot:

- Say "Hello!"
- Store and retrieve values
- Solve math problems
- Make decisions
- Count and control repetitions

These examples don't require interaction with external circuits. In later chapters you will start building circuits and make your robot move. You will also learn additional programming techniques like keeping lists of values and writing pieces of reusable code.

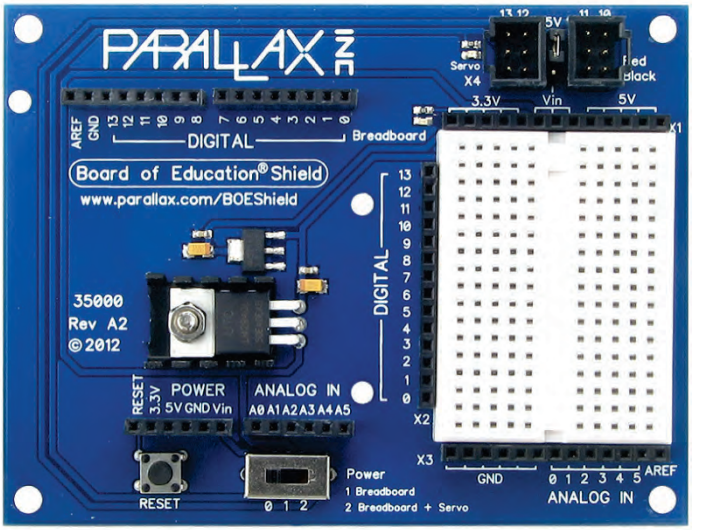
Check Your Hardware

Before continuing, is a good idea to make sure you have all of the correct parts to build and program your Shield-Bot. Use the pictures and part numbers on the following pages to double-check the robot chassis parts, small hardware, and electronic components. If you need anything, contact sales@parallax.com. (Kit parts and quantities subject to change without notice).



Arduino Uno module (#32330). Included in #32335 & #910-32333 only

USB A to B Cable (#805-00007). Included in #32335 only



Parallax Board of Education Shield (#35000)

Shield-Bot Chassis Parts

In Chapter 3, you will build your Shield-Bot on an aluminum chassis. It will use 5 AA batteries for a power supply.

aluminum chassis
(#700-00022)

5-cell AA battery holder with
barrel plug (#753-00007)

Wheels will connect to servo motors to drive the Shield-Bot, and a tail wheel ball will attach to the chassis with a cotter pin. Note that the wheel and tire styles have changed over time. Yours may look different; that's okay.

(2) small robot
wheels and O-ring
tires (#28114)

(2) continuous rotation
servos (#900-00008)

tail wheel
(#700-00009)

Shield-Bot Hardware

A bag of hardware supplies everything you will need to assemble your robot in Chapter 3, shown below. There will be some spare parts left over and not pictured— that is okay! Note that both regular nuts and Nylon-core lock-nuts are provided; you may choose to use either one. Replacement Robot Hardware Refresher Packs (#570-35000) are also sold separately.

(1) 1/16" cotter pin
(#700-00023)

(8) 3/8" 4-40 pan-head screws
(#700-00002)

(2) 3/8" Nylon flat-head screws
(#710-00046)

(8) 1/4" 4-40 pan-head screws
(#700-00028)

(5) 7/8" 4-40 pan-head screws
(#710-00007)

(10) 4-40 steel nuts (#700-00003)

(Nylon-core lock-nuts are also included for optional use (#700-00024)

(4) 1" round 4-40 aluminum standoffs
(#700-00060)

(2) 1/2" round aluminum spacers
(#713-00007)

(1) 3/32" rubber grommet (#700-00025)

(2) #4 Nylon washers (#700-00015)

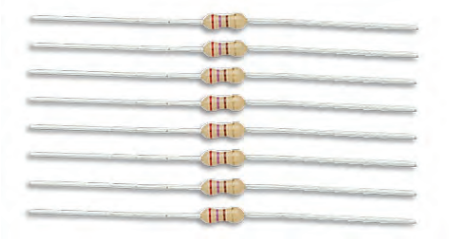
(3) 4-40 Nylon nuts (#700-00036)

(3) 1/2" round Nylon spacers
(#713-00051)

Shield-Bot Electronics Parts

A bag of electronic components is included with your kit, pictured on the next two pages (though parts and quantities are subject to change without notice.) You will use these parts to build circuits in almost every chapter of this book. These parts are available as the Boe-Bot and Shield-Bot Refresher Pack (#572-28132).

Your Infrared Receivers may look different, since suppliers change over time. If you ever need replacements, be sure to order the Infrared Receiver for Boe-Bot and Shield-Bot (#350-00039).



(8) 220 ohm resistors, 1/4 W, 5% (#150-02210)

220 Ω = RED, RED, BROWN

(4) 470 ohm resistors, 1/4 W, 5% (#150-04710)

470 Ω = YELLOW, VIOLET, BROWN

(4) 10 k-ohm resistors, 1/4 W, 5% (#150-01030)

10 kΩ = BROWN, BLACK, ORANGE

(2) 2 k-ohm resistors, 1/4 W, 5% (#150-02020)

2 kΩ = RED, BLACK, RED

(2) 4.7 k-ohm resistors, 1/4 W, 5% (#150-04720)

4.7 kΩ = YELLOW, VIOLET, RED

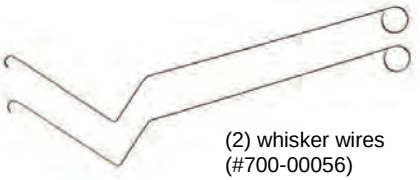
(2) 1 k-ohm resistors, 1/4 W, 5% (#150-01020)

1 kΩ = BROWN, BLACK, RED

(2) bags of 3" jumper wires
(#800-00016)

(2) 3-pin male-male headers
(#451-00303)

(2) tact switches, normally-open pushbuttons (#710-00007)



(2) whisker wires
(#700-00056)

(2) LEDs, red
(#350-00006)

(2) phototransistors
(#350-00029)

(2) LEDs, infrared
(#350-00003)

(2) LED standoffs (#350-90000, use with #350-90001)
-or-
(2) LED shield bases (#350-90004, use with #350-90005)

(2) LED light shields (#350-90001, use with #350-90000)
-or-
(2) LED shield caps (#350-90005, use with #350-90004)

(1) piezospeaker
(#900-000101)

(2) 0.1 µF capacitors
(#200-01040)

(2) 7/8" 0.01 µF capacitors
(#200-01031)

(2) infrared receivers for Boe-Bot or Shield-Bot (#350-00039)

Activity 1: Download Example Sketches and Software

Arduino programs are called *sketches*. All the example sketches in this book can be downloaded from http://learn.parallax.com/shieldrobot. But, if you type them in yourself, you will develop your programming skills faster!

Getting Started with the Arduino IDE Software

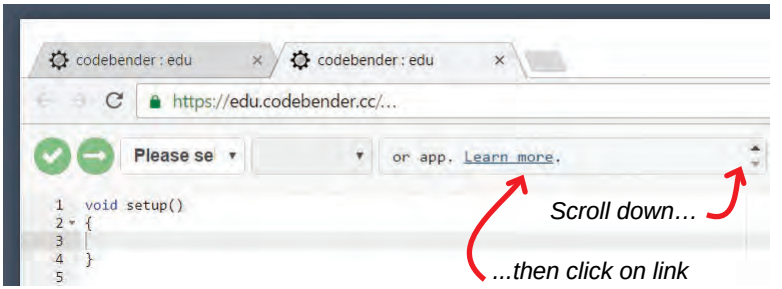
Arduino IDE software and drivers install on your Windows, Mac, or Linux computer. You do not need to be online to use it.

- ✓ Go to www.arduino.cc and click on the Getting Started link.
- ✓ Follow their instructions for downloading and installing the latest Arduino software, and installing the USB driver your system will need to communicate with the Arduino.
- ✓ Make sure to follow the instructions through the part where you connect your Arduino Uno module to your computer with a USB A to B cable, and successfully load a sample sketch to confirm your programming connection.

Getting Started with codebender : edu

Classroom subscriptions are available for the codebender : edu online programming tool. A browser plug-in and drivers are required, and you must be online to use it.

- ✓ Go to https://edu.codebender.cc and scroll down to obtain a subscription.
- ✓ Follow their instructions for registering a new account.
- ✓ From your account URL, find the link to the required browser plug-in options, then select the codebender app for your browser and operating system.

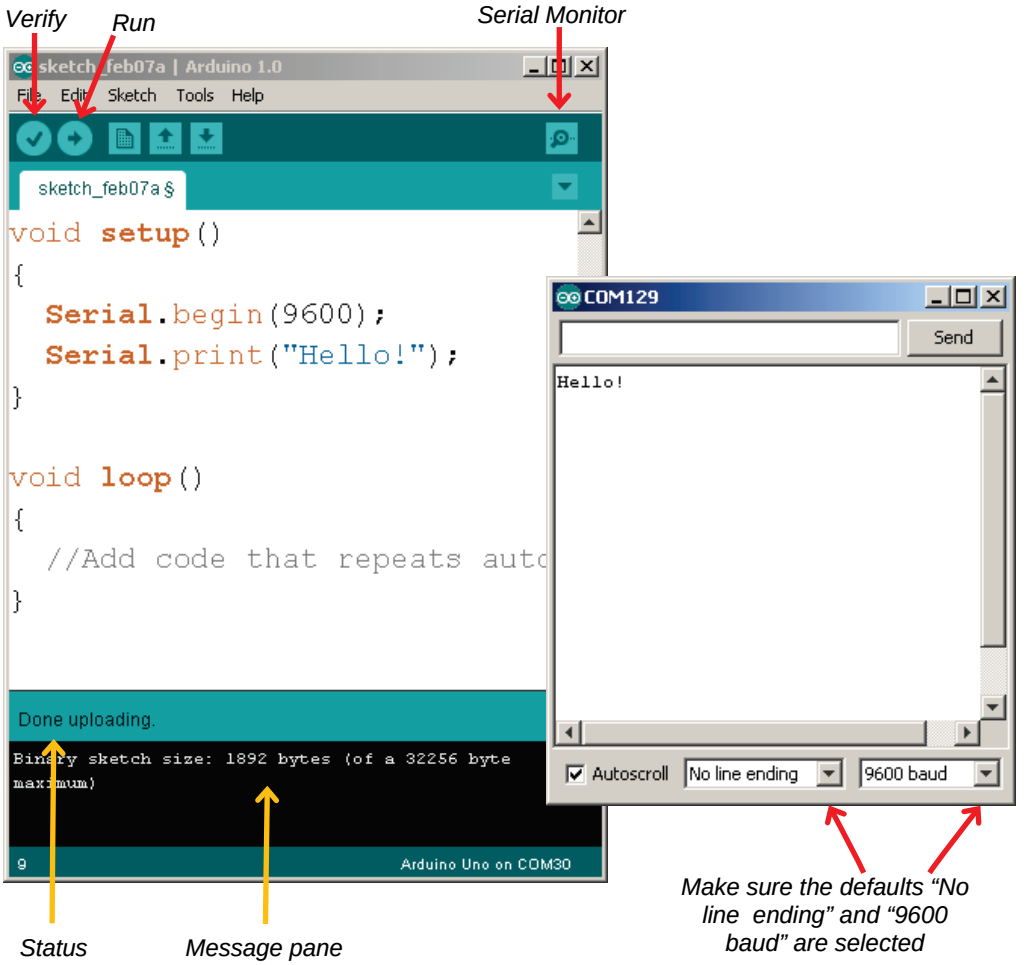


Activity 2: Write a Simple "Hello!" Sketch

Let's start programming! If you are using Arduino IDE, continue below. If you are using codebender : edu, turn to Try it with codebender : edu on page 19.

Try It with Arduino IDE Software

Here is a screen capture of the Arduino Development Environment. The edit pane contains a simple program, called a sketch, that sends a "Hello!" message to the Serial Monitor window on the right.



- ✓ Open your Arduino software and carefully type in the code:

```
void setup()

{
  Serial.begin(9600);
  Serial.print("Hello!");
}

void loop()
{
  //Add code that repeats automatically here.
}
```

- ✓ Be sure you have capitalized "Serial" both times, or the sketch won't work.
- ✓ Also, notice in the figure that the sketch uses parentheses **( )** and curly braces **{}**. Be sure to use the right ones in the right places!
- ✓ Click the Verify button to make sure your code doesn't have any typing errors.
- ✓ Look for the "Binary sketch size" text in the message pane.
- ✓ If it's there, your code compiled and is ready to load to the Arduino.
- ✓ If there's a list of errors instead, it's trying to tell you it can't compile your code. So, find the typing mistake and fix it!
- ✓ Click the arrow button to upload the sketch and run it on the Arduino. The status line under your code will display "Compiling sketch…," "Uploading…," and then "Done uploading."
- ✓ After the sketch is done uploading, click the Serial Monitor button.
- ✓ If the Hello message doesn't display as soon as the Serial Monitor window opens, check for the "9600 baud" setting in the lower right corner of the monitor.
- ✓ Use File → Save to save your sketch. Give it the name HelloMessage.

If you've successfully run the code and verified the output in the Serial Terminal, you are ready to see How the Hello Sketch Code Works on page 20.

Try it with codebender : edu

If you've never used codebender : edu before, take the time to register for a new account. If you are a student, your instructor may have already registered for your class, and would supply a link for your class to share.

- ✓ Take a look at the screen capture of the codebender : edu environment and its Serial Monitor on the next page.

FYI: To keep things simple, after this section, the rest of the Shield Robot tutorials will feature directions and screen captures for the Arduino IDE software.

- ✓ If your Arduino Uno is not already plugged into your computer, do so now.
- ✓ In a browser, go to your codebender : edu page.
- ✓ Type in the following code:

```
void setup()

{
  Serial.begin(9600);
  Serial.print("Hello!");
}

void loop()
{
  // Add code that repeats automatically here.
}
```

- ✓ Be sure you have capitalized "Serial" both times, or the sketch won't work.
- ✓ Also, notice that the sketch uses parentheses **( )** and curly braces **{}**. Be sure to use the right ones in the right places!
- ✓ Verify that the code is written correctly by clicking the Verify button.
- ✓ Choose Arduino Uno (or the board you are using).
- ✓ Select your Arduino's COM port is from the dropdown.
- ✓ Open the Serial Monitor pane by clicking the Serial Monitor button.
- ✓ Once the above steps are completed successfully and your code is verified to be correct, click Run.
- ✓ Your Serial Monitor should be set to 9600 baud , then click the Connect button.
- ✓ Watch the Hello message appear in the Serial Monitor!
- ✓ Click the far right menu and choose Save to download your sketch.

- ✓ Choose Open from this menu to re-open your sketch. Or, navigate to the un-zipped example code archive you downloaded previously to open sketches used in this book.

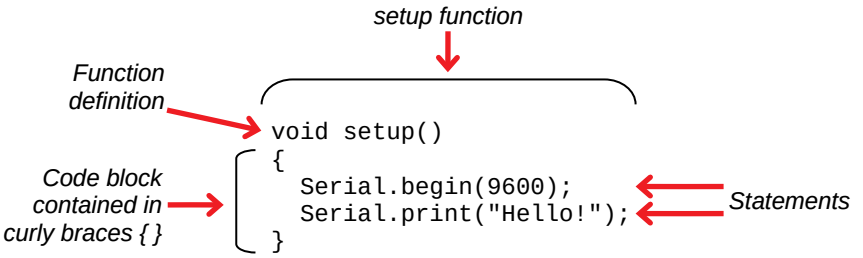
How the Hello Sketch Code Works

A *function* is a container for statements (lines of code) that tell the Arduino to do certain jobs. The Arduino language has two built-in functions: **`setup`** and **`loop`**.
The **`setup`** function is shown below. The Arduino executes the statements you put between the **`setup`** function's curly braces, but only once at the beginning of the sketch.

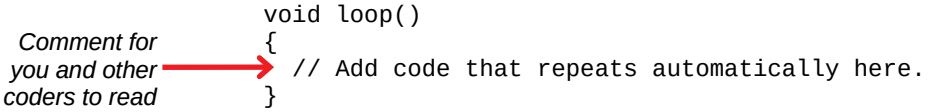
In this example, both statements are *function calls* to functions in the Arduino's built-in, pre-written Serial code library: **`Serial.begin(`*`speed`*`)`** and **`Serial.print(`*`val`*`)`**. Here, ***`speed`*** and ***`val`*** are *parameters*, each describing a value that its function needs passed to it to do its job. The sketch provides these values inside parentheses in each function call.

`Serial.begin(9600);` passes the value 9600 to the ***`speed`*** parameter. This tells the Arduino to get ready to exchange messages with the Serial Monitor at a data rate of 9600 bits per second. That's 9600 binary ones or zeros per second, and is commonly called a *baud rate*.

`Serial.print(val);` passes the message "Hello!" to the ***`val`*** parameter. This tells the Arduino to send a series of binary ones and zeros to the Serial Monitor. The monitor decodes and displays that serial bitstream as the "Hello!" message.



After the **`setup`** function is done, the Arduino automatically skips to the **`loop`** function and starts doing what the statements in its curly braces tell it to do. Any statements in **`loop`** will be repeated over and over again, indefinitely. Since all this sketch is supposed to do is print one "Hello!" message, the **`loop`** function doesn't have any actual commands. There's just a notation for other programmers to read, called a *comment*. Anything to the right of **`//`** on a given line is for programmers to read, not for the Arduino software's compiler. (A *compiler* takes your sketch code and converts it into numbers—a microcontroller's native language.)

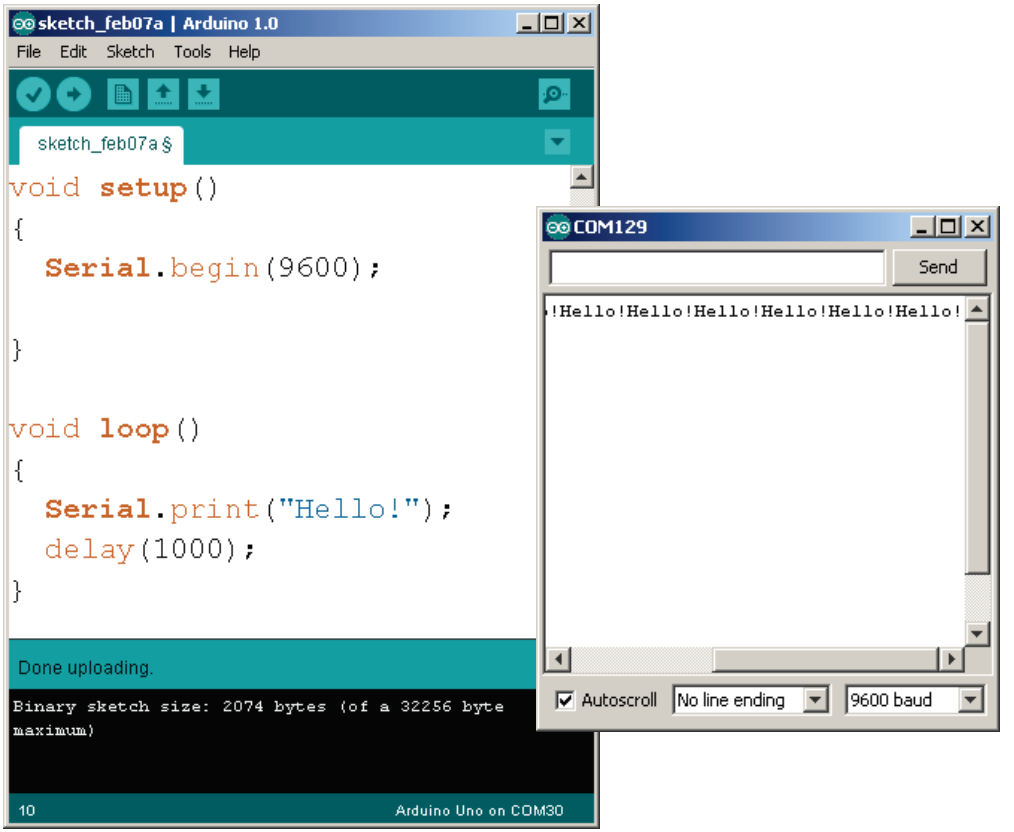


What is `void`? Why do these functions end in **`()`**? The first line of a function is its *definition*, and it has three parts: return type, name, and parameter list. For example, in the function **`void setup`**() the return type is void, the name is setup, and the parameter list is empty – there's nothing inside the parentheses **`()`**. Void means 'nothing'—when another function calls **`setup`** or **`loop`**, these functions would not return a value. An empty parameter list means that these functions do not need to receive any values when they are called to do their jobs.

Modify the Sketch to Repeat

Microcontroller programs generally run in a loop, meaning that one or more statements are repeated over and over again. Remember that the **loop** function automatically repeats any code in its *block* (the statements in between its curly braces).

Let's try moving **Serial.print("Hello!");** to the **loop** function. To slow down the rate at which the messages repeat, let's also add a pause with the built-in **delay(*ms*)** function.



- ✓ Save HelloMessage as HelloRepeated.
- ✓ Move **`Serial.print("Hello!");`** from **`setup`** to the **`loop`** function.
- ✓ Add **`delay(1000);`** on the next line.
- ✓ Compare your changes to the figure and verify that they are correct.
- ✓ Run the sketch on the Arduino and then open the Serial Monitor again.

The added line **`delay(1000)`** passes the value 1000 to the **`delay`** function's ***`ms`*** parameter. It's requesting a delay of 1000 milliseconds. 1 ms is 1/1000 of a second. So, **`delay(1000)`** makes the sketch wait for 1000/1000 = 1 second before letting it move on to the next line of code.

Hello Messages on New Lines

How about having each "Hello!" message on a new line? That would make the messages scroll down the Serial Monitor, instead of across it. All you have to do is change **`print`** to **`println`**, which is short for 'print line.'

- ✓ Change **`Serial.print("Hello!")`** to **`Serial.println("Hello!")`**.
- ✓ Run the modified sketch and watch it print each "Hello!" message on a new line.

Open the Arduino Reference

Still have questions? Try the Arduino Language Reference. It's a set of pages with links you can follow to learn more about **setup**, **loop**, **print**, **println**, **delay**, and lots of other functions you can use in your sketches.

- ✓ If you are using the Arduino IDE, click Help and select Reference.
- ✓ If you are using codebender : edu, go to the following web address: https://www.arduino.cc/en/Reference/HomePage

Try looking up whichever term you might have a question about. You'll find **setup**, **loop**, and **delay** on the main reference page.

If you're looking for links to **print** or **println**, you'll have to find and follow the **Serial** link first.

Activity 3: Store and Retrieve Values

Variables are names you can create for storing, retrieving, and using values in the Arduino microcontroller's memory. Here are three example *variable declarations* from the next sketch:

```
int a = 42;
char c = 'm';
float root2 = sqrt(2.0);
```

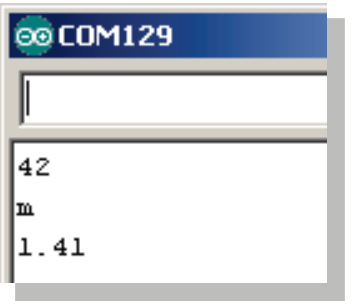
The declaration **int a = 42** creates a variable named **a**. The **int** part tells the Arduino software what *type* of variable (or *data type*) it's dealing with. The **int** type can store integer values ranging from -32,768 to 32,767. The declaration also assigns **a** an initial value of 42. (The initial value is optional, you could instead just declare **int a**, and then later assign the value 42 to a with **a = 42**.)

Next, **char c = 'm'** declares a variable named **c** of the type **char** (which is for storing characters) and then assigns it the value 'm'.

Then, **float root2 = sqrt(2.0)** declares a variable named **root2**. The variable type is **float**, which can hold decimal values. Here, **root2** is initialized to the floating-point representation of the square root of two: **sqrt(2.0)**.

Now that your code has stored values to memory, how can it retrieve and use them? One way is to simply pass each variable to a function's parameter. Here are three examples, where the **Serial.println(*val*)** function displays the value of the variable inside the parentheses.

```
Serial.println(a);
Serial.println(c);
Serial.println(root2);
```



One nice thing about variable types is that **`Serial.println`** recognizes each type and displays it correctly in the Serial Monitor. (Also, the C++ compiler in the Arduino software requires all declared variables to have a type, so you can't leave it out.)

Example Sketch – StoreRetrieveLocal

- ✓ Create a new sketch, and save it as StoreRetrieveLocal.
- ✓ Open or create and save the StoreRetrieveLocal sketch, and run it on your Arduino.
- ✓ Open the Serial Monitor and verify that the values display correctly.

```
// Robotics with the BOE Shield - StoreRetrieveLocal

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

  int a = 42;
  char c = 'm';
  float root2 = sqrt(2.0);

  Serial.println(a);
  Serial.println(c);
  Serial.println(root2);
}

void loop()
{
  // Empty, no repeating code.
}
```

ASCII stands for American Standard Code for Information Exchange. It's a common code system for representing computer keys and characters in displays. For example, the declaration **`char c = 'm'`** makes the Arduino store the number 109 in the **`c`** variable. **`Serial.println(c)`** makes the Arduino send the number 109 to the Serial Monitor. When the Serial Monitor receives that 109, it automatically displays the letter m. See http://learn.parallax.com/ascii for codes 1-127.

See that 'm' really is 109!

There are two ways to prove that the ASCII code for 'm' really is 109. First, instead of declaring **`char c = 'm'`**, you could use **`byte c = 'm'`**. Then, the **`println`** function will print the **`byte`** variable's decimal value instead of the character it represents. Or, you could leave the **`char c`** declaration alone and instead use **`Serial.println(c, DEC)`** to display the decimal value **`c`** stores.

- ✓ Try both approaches. So, do you think the letters l, m, n, o, and p would be represented by the ASCII codes 108, 109, 110, 110, 111, and 112?
- ✓ Modify your sketch to find out the decimal ASCII codes for l, m, n, o, p.
- ✓ If you can, go to http://learn.parallax.com/ascii and try other ASCII characters.

Global vs.Local Variables

So far, we've declared variables inside a function block (inside the function's curly braces), which means they are *local variables*. Only the function declaring a local variable can see or modify it. Also, a local variable only exists while the function that declares it is using it. After that, it gets returned to unallocated memory so that another function (like **`loop`**) could use that memory for a different local variable.

If your sketch has to give more than one function access to a variable's value, you can use *global variables*. To make a variable global, just declare it outside of any function, preferably before the **`setup`** function. Then, all functions in the sketch will be able to modify or retrieve its value. The next example sketch declares global variables and assigns values to them from within a function.

Example Sketch – StoreRetrieveGlobal

This example sketch declares **`a`**, **`c`**, and **`root2`** as global variables (instead of local). Now that they are global, both the **`setup`** and **`loop`** functions can access them.

- ✓ Modify your sketch to match the one below, and save it as StoreRetrieveGlobal.
- ✓ Run it on the Arduino, and then open the Serial Monitor and verify that the correct values are displayed repeatedly by the **`loop`** function.

```
// Robotics with the BOE Shield - StoreRetrieveGlobal

int a;
char c;
float root2;

void setup()
{
  Serial.begin(9600);
```

```
  a = 42;
  c = 'm';
  root2 = sqrt(2.0);
}

void loop()
{
  Serial.println(a);
  Serial.println(c);
  Serial.println(root2);
  delay(1000);
}
```

Your Turn – More Data Types

There are lots more data types than just **int**, **char**, **float**, and **byte**.

- ✓ Open the Arduino Language Reference, and check out the Data Types list.
- ✓ Follow the **float** link and learn more about this data type.
- ✓ The **long** data type will be used in a later chapter; open both the **long** and **int** sections. How are they similar? How are they different?

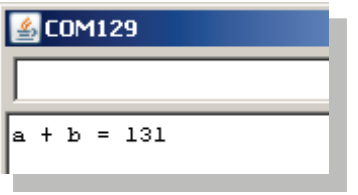
Activity 4: Solve Math Problems

Arithmetic operators are useful for doing calculations in your sketch. In this activity, we'll focus on the basics: assignment (**=**), addition (**+**), subtraction (**-**), multiplication (*****), division (**/**), and modulus (**%**, the remainder of a division calculation).

- ✓ Open up the Arduino Language Reference, and take a look at the list of Arithmetic Operators.

The next example sketch, SimpleMath, adds the variables **a** and **b** together and stores the result in **c**. It also displays the result in the Serial Monitor.

Notice that **c** is now declared as an **int**, not a **char** variable type. Another point, **int c = a + b** uses the assignment operator **(=)** to copy the result of the addition operation that adds **a** to **b**. The next screen capture shows the expected result of 89 + 42 = 131 in the Serial Monitor.



- ✓ Create, save, and run SimpleMath on your Arduino.
- ✓ Check the result in the Serial Monitor. Is it correct?

```
// Robotics with the BOE Shield - SimpleMath

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

  int a = 89;
  int b = 42;
  int c = a + b;

  Serial.print("a + b = ");
  Serial.println(c);
}

void loop()
{
  // Empty, no repeating code.
}
```

Fit your variables to the result values you need to store. This will use less memory so you can write larger sketches that will execute more efficiently.

- If you need to work with decimal point values, use **`float`**.
- If you are using integer values (counting numbers), choose **`byte`**, **`int`**, or **`long`**.
- If your results will always be an unsigned number from 0 to 255, use **`byte`**.
- If your results will not exceed –32,768 to 32,767, an **`int`** variable works.
- If you need a larger range of values, try a **`long`** variable instead. It can store values from -2,147,483,648 to 2,147,483,647.

Your Turn – Experiment with Other Arithmetic Operators

You still have **–**, **`*`**, **`/`**, and **`%`** to try out!

- ✓ Replace the addition (**+**) operator with the subtraction (**–**) operator. Replace both instances of **+** in the sketch.
- ✓ Run the modified sketch and verify the result in the Serial Monitor.
- ✓ Repeat for the multiplication (**`*`**), division (**`/`**) and modulus (**`%`**) operators.

Floating Point Math

Imagine entering your BOE Shield-Bot in a contest where you have to make it travel in a circle, but the radius will only be announced a few minutes before the contest. You'd have an advantage if your code could calculate the circumference of the circle. The formula is 2 × π ×

r, where r is the circle's radius and π ≈ 3.14159. This calculation would be a lot easier to do with floating point math.

Here is a snippet of code that gets the job done. Notice that it uses PI instead of 3.14159. **PI** is a built-in C language *constant* (a named value that does not change throughout the sketch). Also notice that all the values have decimal points. That makes them all floating-point values.

```
  float r = 0.75;
  float c = 2.0 * PI * r;
```

Example Sketch - Circumference

- ✓ Create the Circumference sketch and save it.
- ✓ Make sure to use the values 0.75 and 2.0. Do not try to use 2 instead of 2.0.
- ✓ Run your sketch on the Arduino and check the results with the Serial Monitor.

```
// Robotics with the BOE Shield - Circumference

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

  float r = 0.75;
  float c = 2.0 * PI * r;

  Serial.print("circumference = ");
  Serial.println(c);
}

void loop()
{
  // Empty, no repeating code.
}
```

Your Turn – Circle Area

The area of a circle is a = π × r2. Hint: r2 can be expressed as simply r × r.

- ✓ Save your sketch as CircumferenceArea.
- ✓ Add another **float** variable to store the result of an area calculation, and code to display it. Run the sketch, and compare the answer to a calculator's answer.

Activity 5: Make Decisions

Your BOE Shield-Bot will need to make a lot of navigation decisions based on sensor inputs. Here is a simple sketch that demonstrates decision-making. It compares the value of **a** to **b**, and sends a message to tell you whether or not **a** is greater than **b**, with an **if…else** statement.

If the condition **(a > b)** is true, it executes the **if** statement's code block: **Serial.print("a is greater than b")**. If **a** is *not* greater than **b**, it executes the **else** code block instead: **Serial.print("a is not greater than b")**.

- ✓ Create the SimpleDecisions sketch, save it, and run it on the Arduino.
- ✓ Open the Serial Monitor and test to make sure you got the right message.
- ✓ Try swapping the values for **a** and **b**.
- ✓ Re-load the sketch and verify that it printed the other message.

```
// Robotics with the BOE Shield - SimpleDecisions

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

  int a = 89;
  int b = 42;

  if(a > b)
  {
    Serial.print("a is greater than b");
  }
  else
  {
    Serial.print("a is not greater than b");
  }
}

void loop()
{
  // Empty, no repeating code.
}
```

More Decisions with if... else if

Maybe you only need a message when **a** is greater than **b**. If that's the case, you could cut out the **else** statement and its code block. So, all your **setup** function would have is the one **if** statement, like this:

```
void setup()
{
  Serial.begin(9600);

  int a = 89;
  int b = 42;

  if(a > b)
  {
    Serial.print("a is greater than b");
  }
}
```

Maybe your sketch needs to monitor for three conditions: greater than, less than, or equal. Then, you could use an **`if…else if…else`** statement.

```
if(a > b)
  {
    Serial.print("a is greater than b");
  }
  else if(a < b)
  {
    Serial.print("a is not greater than b");
  }
  else
  {
    Serial.print("a is equal to b");
  }
```

A sketch can also have multiple conditions with the Arduino's *Boolean operators*, such as **&&** and **||**. The **&&** operator means AND; the **||** operator means OR. For example, this statement's block will execute only if **a** is greater than 50 AND **b** is less than 50:

```
 if((a > 50) && (b < 50))
  {
    Serial.print("Values in normal range");
  }
```

This example prints the warning message if **a** is greater than 100 OR **b** is less than zero.

```
if((a > 100) || (b < 0))
  {
    Serial.print("Danger Will Robinson!");
  }
```

One last example: if you want to make a comparison to find out if two values are equal, you have to use two equal signs next to each other: **==**.

```
if(a == b)
  {
    Serial.print("a and b are equal");
  }
```

- ✓ Try these variations in a sketch.

More conditions: You can chain more **`else if`** statements after the initial **`if`**.
- The example in this activity only uses one **`else if`**, but you could use more.
- The rest of the statement gets left behind after it finds a true condition.
- If the if statement turns out to be true, its code block gets executed and the rest of the chain of **`else ifs`** gets passed by.

Activity 6: Count and Control Repetitions

Many robotic tasks involve repeating an action over and over again. Next, we'll look at two options for repeating code: the **`for`** loop and **`while`** loop. The **`for`** loop is commonly used for repeating a block of code a certain number of times. The **`while`** loop is used to keep repeating a block of code as long as a condition is true.

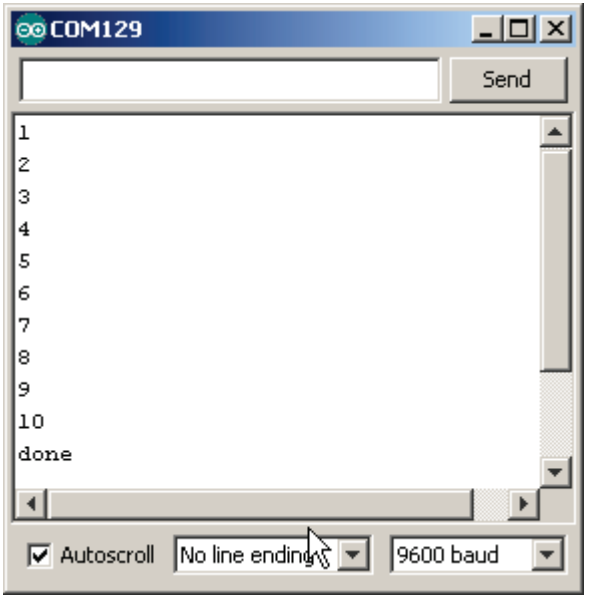
A for Loop is for Counting

A **`for`** loop is typically used to make the statements in a code block repeat a certain number of times. For example, your BOE Shield-Bot will use five different values to make a sensor detect distance, so it needs to repeat a certain code block five times. For this task, we use a **`for`** loop. Here is an example that uses a **`for`** loop to count from 1 to 10 and display the values in the Serial Monitor.

- ✓ Create and save the CountToTen sketch, and run it on your Arduino.
- ✓ Open the Serial Monitor and verify that it counted from one to ten.

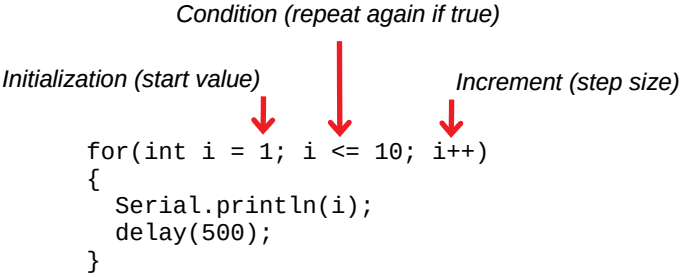
```
// Robotics with the BOE Shield - CountToTen

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

  for(int i = 1; i <= 10; i++)
  {
    Serial.println(i);
    delay(500);
  }
  Serial.println("All done!");
}

void loop()
```

```
{
  // Empty, no repeating code.
}
```



How the for Loop Works

The figure below shows the **for** loop from the last example sketch, CountTenTimes. It labels the three elements in the **for** loop's parentheses that control how it counts.



Initialization: the starting value for counting. It's common to declare a local variable for the job as we did here with **int i = 1;** naming it **i** for 'index.'

Condition: what the **for** loop checks between each repetition to make sure the condition is still true. If it's true, the loop repeats again. If not, it allows the code to move on to the next

statement that follows the **for** loop's code block. In this case, the condition is "if **i** is less than or equal to 10."

Increment: how to change the value of **i** for the next time through the loop. The expression **i++** is equivalent to **i = i + 1**. It makes a nice shorthand approach for adding 1 to a variable. Notice that **++** comes after **i**, meaning "use **i** as-is this time through the function, and then increment it afterward." This is the *post increment* use of the operator.

The first time though the loop, the value of **i** starts at 1. So, **Serial.println(i)** displays the value 1 in the Serial Monitor. The next time through the loop, **i++** has made the value of **i** increase by 1. After a **delay** (so you can watch the individual values appear in the Serial Monitor), the **for** statement checks to make sure the condition **i <= 10** is still true. Since **i** now stores 2, it is true since 2 is less than 10, so it allows the code block to repeat again. This keeps repeating, but when **i** gets to 11, it does not execute the code block because it's not true according to the **i <= 10** condition.

Adjust Initialization, Condition, and Increment

As mentioned earlier, **i++** uses the **++** increment operator to add 1 to the **i** variable each time through the **for** loop. There are also compound operators for decrement **--**, and compound arithmetic operators like **+=**, **-=**, ***=**, and **/=**. For example, the **+=** operator can be used to write i = i + 1000 like this: **i+=1000**.

- ✓ Save your sketch, then save it as CountHigherInSteps.
- ✓ Replace the **for** statement in the sketch with this:

```
for(int i = 5000; i <= 15000; i+=1000)
```

- ✓ Run the modified sketch and watch the output in the Serial Monitor.

A Loop that Repeats While a Condition is True

In later chapters, you'll use a **while** loop to keep repeating things while a sensor returns a certain value. We don't have any sensors connected right now, so let's just try counting to ten with a **while** loop:

```
int i = 0;
  while(i < 10)
  {
    i = i + 1;
    Serial.println(i);
    delay(500);
  }
```

Want to condense your code a little? You can use the increment operator (**++**) to increase the value of **i** inside the **Serial.println** statement. Notice that **++** is on the left side of the **i** variable in the example below. When **++** is on the left of a variable, it adds 1 to the value of **i** *before* the **println** function executes. If you put **++** to the right, it would add 1 *after* **println** executes, so the display would start at zero.

```
int i = 0;
 while(i < 10)
 {
   Serial.println(++i);
   delay(500);
 }
```

The **loop** function, which must be in every Arduino sketch, repeats indefinitely. Another way to make a block of statements repeat indefinitely in a loop is like this:

```
int i = 0;
 while(true)
 {
   Serial.println(++i);
   delay(500);
}
```

So why does this work? A **while** loop keeps repeating as long as what is in its parentheses evaluates as true. The word 'true' is actually a pre-defined constant, so **while(true)** is always true, and will keep the **while** loop looping. Can you guess what **while(false)** would do?

- ✓ Try these three different **while** loops in place of the **for** loop in the CountToTen sketch.
- ✓ Also try one instance using **Serial.println(++i)**, and watch what happens in the Serial Monitor.
- ✓ Try **while(true)** and **while(false)** also.

Activity 7: Constants and Comments

The next sketch, CountToTenDocumented, is different from CountToTen in several ways. First, it has a block comment at the top. A *block comment* starts with **/*** and ends with ***/**, and you can write as many lines of notes in between as you want. Also, each line of code has a *line comment* (starting with //) to its right, explaining what the code does.

Last, three **const int** (constants that are integers) are declared at the beginning of the sketch, giving the names **startVal**, **endVal**, and **baudRate** to the values 1, 10, and 9600. Then, the sketch uses these names wherever it requires these values.

```
/*
 Robotics with the BOE Shield – CountToTenDocumented
 This sketch displays an up-count from 1 to 10 in the Serial Monitor
 */

const int startVal = 1;                       // Starting value for counting
const int endVal = 10;                        // Ending value for counting
const int baudRate = 9600;                    // For setting baud rate

void setup()                                  // Built in initialization block
{
  Serial.begin(baudRate);                     // Set data rate to baudRate

  for(int i = startVal; i <= endVal; i++)     // Count from startVal to endVal
  {
    Serial.println(i);                        // Display i in Serial Monitor
    delay(500);                               // Pause 0.5 s between values
  }
  Serial.println("All done!");                // Display message when done
}

void loop()                                   // Main loop auto-repeats
{
  // Empty, no repeating code.
}
```

Documenting Code

Documenting code is the process of writing notes about what each part of the program does. You can help make your code self-documenting by picking variable and constant names that help make the program more self-explanatory. If you are thinking about working in a field that involves programming, it's a good habit to start now. Why?

- Folks who write code for a living, like software developers and robotics programmers, are usually under orders to document their code.
- Other people might need to make updates to your code or use it for another project, and they need to understand what the code does.

- Documented code can save you lots of time trying to remember what your code does, and how it does it, after you haven't looked at it for a long time.
- In addition to making your code easier to read, constants allow you to adjust an often-used value quickly and accurately by updating a single constant declaration. Trying to find and update each instance of an unnamed value by hand is an easy way to create bugs in your sketch!

Now it is your turn to give it a try, and develop a new, good habit.

- ✓ Read through the sketch to see how constants and comments are used.
- ✓ Open up the SimpleDecisions sketch and document it, using the sketch CountToTenDocumented as an example.
- ✓ Add a title and detailed description of the sketch, enclosed in a block comment.
- ✓ Use **`const`** declarations to name the values of 89 and 42.
- ✓ Use the names from your **`const`** declarations instead of 89 and 42 in the **`setup`** function.
- ✓ Add line comments to the right of each line.

Chapter 1 Summary

After going to the Arduino site to install and test your software and programming connection, this chapter guided you through several programming activities. These example sketches showed you how to make your microcontroller do some common tasks, introduced many programming concepts, and suggested a couple of good habits to develop your computer skills.

Microcontroller Tasks

- Sending messages to a serial monitor for display
- Storing and retrieving values from memory
- Solving math problems
- Making decisions to control program flow
- Counting and controlling repetitions

Programming Concepts

- What a function is, and how to pass a value to a function's parameter
- What the Arduino's **`setup`** and **`loop`** functions do
- The difference between global and local variables
- Declaring and using variable data types, with examples of **`char`**, **`int`**, and **`float`**
- How to solve math problems with arithmetic operators
- How to make decisions with **`if`**, **`if…else`**, and **`if…else if`**

- Using operators in **`if`** statements
- How to count and control repetitions with **`for`** and **`while`** loops

Computer Skills

- What a baud rate is, and how to set it in your sketch and your Serial Monitor
- What ASCII characters are, and what they are used for
- Using your microcontroller's language reference
- Why documenting code is important, and how to help make your code self-documenting

Chapter 1 Challenges

Questions

1. What device will be the brain of your BOE Shield-Bot?
2. When the Arduino sends a character to your computer, what type of numbers are used to send the message through the programming cable?
3. What is the difference between the **`setup`** and **`loop`** functions?
4. What's the difference between a variable name and a variable type?
5. What's the difference between global and local variables?
6. What are the arithmetic operators? What does each one do?
7. What variable type will the Arduino editor apply to 21.5 if it appears in your code? Why?
8. What three elements are included between parentheses in a **`for`** loop?
9. What's the difference between a block comment and a line comment?

Exercises

1. Write a piece of code that displays "the value of i = " followed by the value of stored in the variable **`i i`**n the Serial Monitor. Both should display on the same line, and then move the cursor to the beginning of the next line for displaying more messages.
2. Declare a **`long`** variable named **`bigVal`**, and initialize it to 80 million.
3. Write an **`if…else`** statement that takes the modulus of a variable divided by 2 and compares it to zero. If the result is zero, display "The variable is even." If not, display "The variable is odd."
4. Write a **`for`** loop that starts counting at 21 and stops at 39, and counts in steps of 3.
5. Write a piece of code that displays the character stored by a variable stores along with its ASCII value.
6. Write a **`for`** loop, but instead of counting from one value to another, make it count from 'A' to 'Z' and display the letters in the alphabet.

Projects

1. Write a sketch to display the printable ASCII characters. The first printable character is the space character, which is one press/release of your keyboard's space bar between apostrophes, like this: ' '. The last printable character is the tilde character '~'. Alternately, you could use 32 for the loop's start value and 126 for the end value.
2. Write a sketch that tells you if a variable is odd or even. Hint: when a number is even, the remainder of the number divided by 2 is 0. Hint: *variable % 2 == 0*.

Question Solutions

1. The Arduino module.
2. Binary numbers, that is, 0's and 1's. We also saw examples of how the numbers that represent characters are ASCII codes, like 109 = 'm'.
3. The **`setup`** function's statements get executed once when the sketch starts. After finishing the **`setup`** function, the sketch advances to the **`loop`** function. Its code block gets repeated indefinitely.
4. The variable's name is used for assignment and comparison in the sketch. The variable's type defines the kind and range of values it can store.
5. Global variables can be accessed and modified by any function in the sketch. Local variables can only be accessed and modified within the block where they are declared.
6. The arithmetic operators are **+** add, **-** subtract, ***** multiply, **/** divide, and **%** modulus.
7. It will be treated as a **`float`** type because it has a decimal point.
8. Initialization, condition, increment.
9. A block comment starts with **`/*`** and ends with **`*/`**, and allows you to write comments that span multiple lines. A line comment starts with **`//`** and makes whatever is to its right on that particular line a comment.

Exercise Solutions

1. Solution:

```
Serial.print("the value of i = ");
Serial.println(i);
```

2. Solution:

```
long bigVal = 80000000;
```

3. Solution:

```
if(myVar % 2 == 0)
{
  Serial.println("The variable is even. ");
}
else
{
  Serial.println("The variable is odd. ");
}
```

4. Solution:

```
for(int i = 21; i <= 39; i+=3)
{
  Serial.print("i = ");
  Serial.println(i);
}
```

5. Solution:

```
char c = "a";
Serial.print("Character = ");
Serial.print(c);
Serial.print("   ASCII value = ");
Serial.println(c, DEC);
```

6. Solution:

```
for(char c = 'A'; c <='Z'; c++){}
```

Project Solutions

1. This sketch is a modified version of CountToTen that utilizes the solution to Exercise 5 for displaying ASCII characters.

```
// Robotics with the BOE Shield - Chapter 1, Project 1

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

  for(char c = ' '; c <= '~'; c++)
  {
    Serial.print("Character = ");
    Serial.print(c);
    Serial.print("   ASCII value = ");
    Serial.println(c, DEC);
```

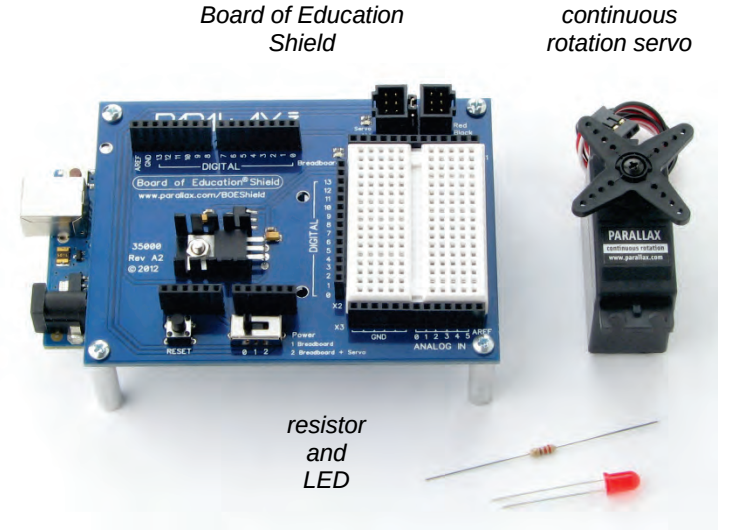
```
  }
  Serial.println("All done!");
}

void loop()
{
  // Empty, no repeating code.
}
```

2. This sketch is a modified version of SimpleDecisions that uses a variation of the solution from Exercise 3 to display whether the variable is odd or even.

```
// Robotics with the BOE Shield - Chapter 1, Project 2

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

  int a = 20;

  if(a % 2 == 0)
  {
    Serial.print("a is even");
  }
  else
  {
    Serial.print("a is odd");
  }
}

void loop()
{
  // Empty, no repeating code.
}
```

Chapter 2. Shield, Lights, Servo Motors

In this chapter, you will use the Board of Education Shield for building and testing circuits with Parallax continuous rotation servos, resistors, and LED lights. Along the way, you'll start learning the basics of building circuits and making the Arduino interact with them. By the end of the chapter, you'll have a pair of servos connected, each with its own signal indicator light, and you'll be writing sketches to control servo speed and direction.



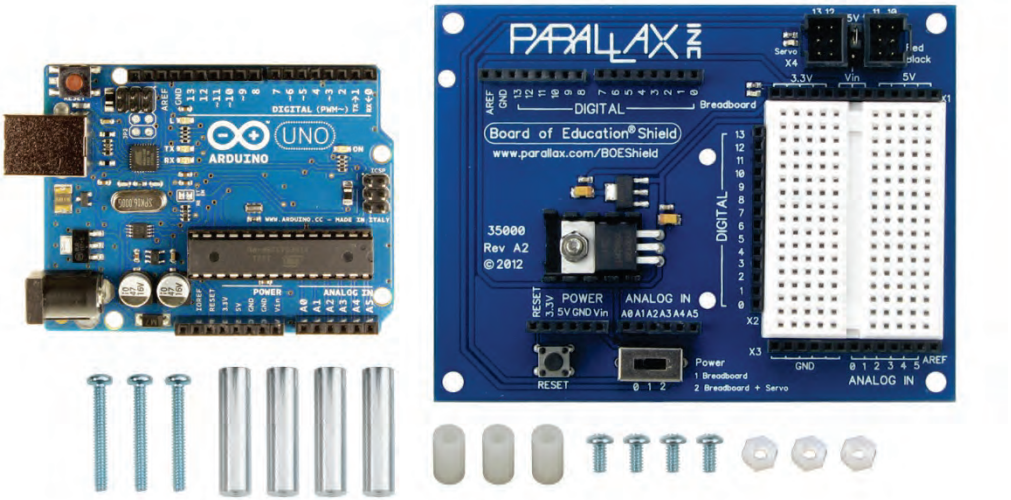
Activity 1: Board of Education Shield Setup

The Board of Education Shield makes it easy to build circuits and connect servos to the Arduino module. In this chapter, you will use it to test servos and indicator lights. Next chapter, you'll mount the BOE Shield and servos on a robot chassis to build a robot we'll call the BOE Shield-Bot.

Parts List

(1) Arduino module
(1) Board of Education Shield
(4) 1″ round aluminum standoffs
(4) pan head screws, 1/4″ 4-40 (metal, and rounded on top with a +)
(3) 1/2″ round Nylon spacers*
(3) Nylon nuts, 4-40*
(3) pan head screws, 7/8″, 4-40*

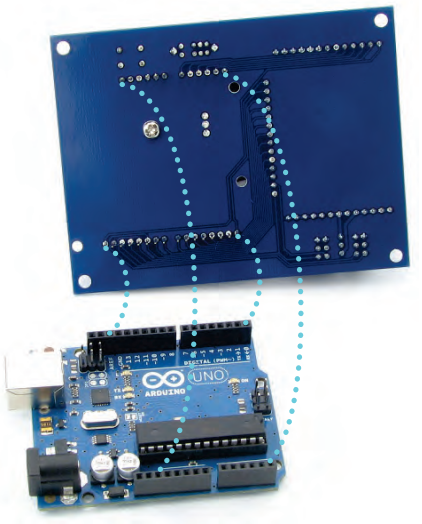
(*Items also included in the Boe-Bot to Shield-Bot Retrofit Kit, (#32333).



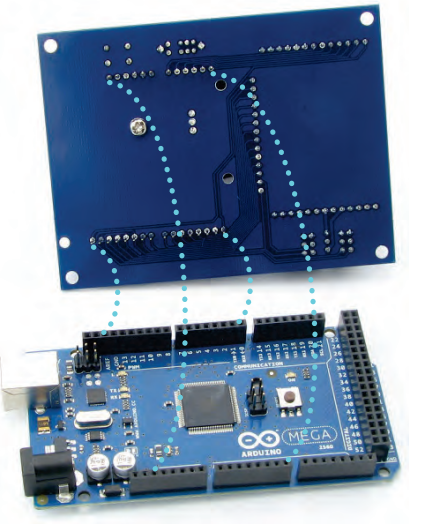
Instructions

The four groups of pins under the Board of Education Shield plug into the four Arduino socket headers. There are also three board-connection holes in the shield that line up with holes in the Arduino module, designed to connect the two boards together with screws and Nylon spacers.

If you have a revision 3 Arduino, it will be labeled UNO R3 or MEGA R3 on the back. R3 boards will have two empty pairs of sockets, closest to the USB and power connectors, after socketing the shield. Earlier versions, such as 2, 1, and Duemilanove, have the same number of sockets as the shield has pins, so there will be no empty sockets left over. If you have an Arduino Mega, the four pin groups will fit into the four headers closest to the USB and power connectors, as shown next.



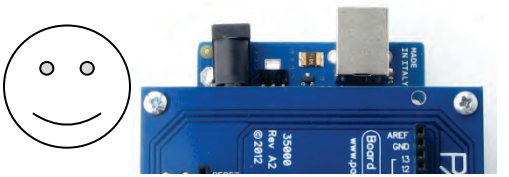
Uno R3

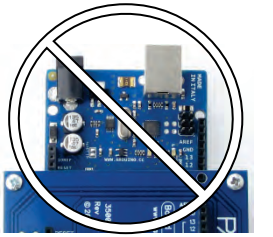


Mega R3

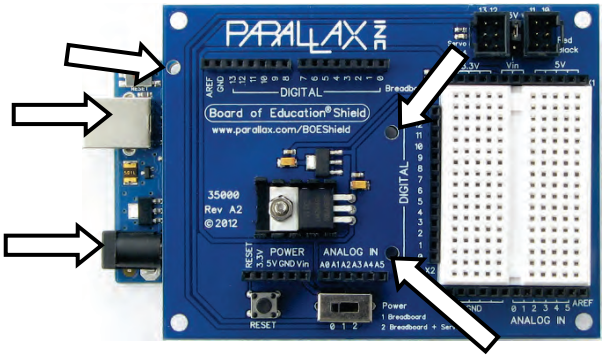
- ✓ Disconnect the programming cable from your Arduino module.
- ✓ Remove the foam covering the metal pins on the bottom of the Board of Education Shield.

Look closely at your Arduino module and the pins on the Board of Education Shield to see how the sockets and pins will line up for your particular boards. Note that if you have an Arduino Mega, its USB port and power jack will be close to the edge of the shield, like the image above-right, and below-left.

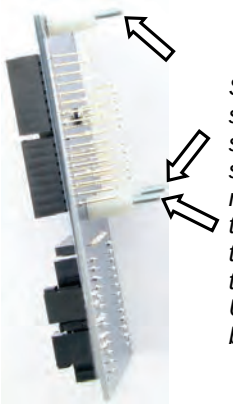




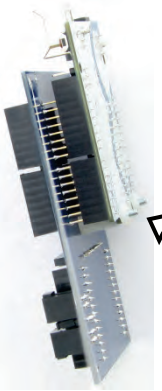
Component placement varies a little bit for the different Arduino models; some can only fit one or two Nylon standoffs for holding the boards together. This is okay, but you need to find out which holes you can use before socketing the Board of Education Shield.



- ✓ Hold a Nylon spacer over each mounting hole on your Arduino module, and look through it to see if the spacer can line up with the hole completely. For the mounting hole by the Arduino Uno reset button, a spacer may not fit.
- ✓ Insert a 7/8″ screw through the corresponding board-connection holes in your Board of Education Shield.



Slide Nylon spacers over screws. A spacer may not fit with the screw that goes by the Arduino Uno's reset button.



After socketing shield, thread and tighten Nylon nuts

- ✓ Slide a Nylon spacer over each screw you used in mounting holes where a spacer will fit. It is okay to use the screw without the spacer where needed.
- ✓ Line up the Arduino module's sockets with the Board of Education Shield's pins.
- ✓ Also line up the 7/8″ screws with the mounting holes in the Arduino board.
- ✓ Gently press the two boards together until the pins are firmly seated in their sockets. The sockets will not cover the pins completely; there will be about 3/8″ (~5 mm) of the pins still exposed between the bottom of the shield and the top of the sockets.

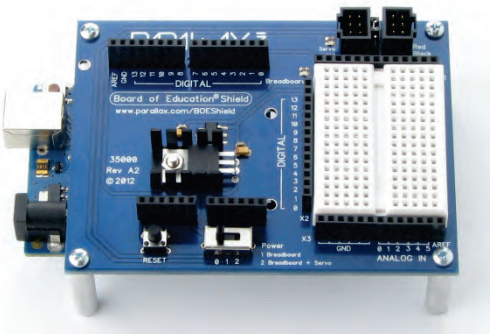
- ✓ Check to make ABSOLUTELY SURE your pins are seated in the sockets correctly. It is possible to misalign the pins, which can damage your board when it is powered.

Correct: gap in pins lines up with gap in sockets.

WRONG! STOP! There's a pin between the gap in the sockets. Unplug it and try again.

- ✓ Thread a Nylon nut over each screw, and tighten gently.

To keep the connected boards up off of the table, we'll mount tabletop standoffs to each corner of the Board of Education Shield.



- ✓ Thread a 1/4″ steel pan-head screw through a corner hole on the Board of Education Shield from the top side, and secure it with a 1" aluminum standoff.
- ✓ Repeat until all four standoffs are installed.

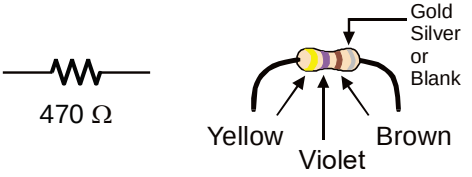
Activity 2: Build and Test LED Indicator Lights

Indicator lights give people a way to see a representation of what's going on inside a device, or patterns of communication between two devices. Next, you will build indicator lights to display the communication signals that the Arduino will send to the servos. If you haven't ever built a circuit before, don't worry, this activity shows you how.

Introducing the Resistor

A *resistor* is a component that resists the flow of electricity. This flow of electricity is called *current*. Each resistor has a value that tells how strongly it resists current flow. This resistance value is called the *ohm*. The sign for the ohm is the Greek letter omega: Ω. (Later on you will see the symbol kΩ, meaning kilo-ohm, which is one thousand ohms.)

This resistor has two wires (called *leads* and pronounced "leeds"), one coming out of each end. The ceramic case between the two leads is the part that resists current flow. Most circuit diagrams use the jagged line symbol with a number label to indicate a resistor of a certain value, a 470 Ω resistor in this case. This is called a *schematic symbol*. The part drawing on the right is used in some beginner-level texts to help you identify the resistors in your kit, and where to place them when you build circuits.



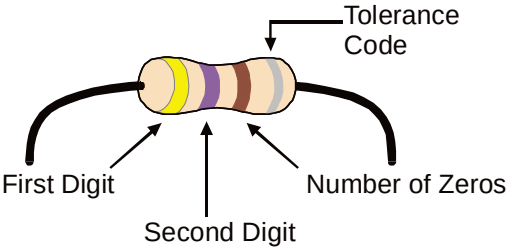
The resistors in your parts kit have colored stripes that indicate what their resistance values are. There is a different color combination for each resistance value. For example, the color code for the 470 Ω resistor is yellow-violet-brown.

There may be a fourth stripe that indicates the resistor's tolerance. *tolerance* is measured in percent, and it tells how far off the part's true resistance might be from the labeled resistance. The fourth stripe could be gold (5%), silver (10%) or no stripe (20%). For the activities in this book, a resistor's tolerance does not matter, but its value does.

Resistor Color Code Values

Each color bar on the resistor's case corresponds to a digit, as listed in the table below.

Digit	0	1	2	3	4	5	6	7	8	9
Color	black	brown	red	orange	yellow	green	blue	violet	gray	white



Here's how to find the resistor's value, in this case proving that yellow-violet-brown is really 470 Ω:

1. The first stripe is yellow, which means the leftmost digit is a 4.
2. The second stripe is violet, which means the next digit is a 7.
3. The third stripe is brown. Since brown is 1, it means add one zero to the right of the first two digits.

Yellow-Violet-Brown = 4-7-0 = 470 Ω.

Your Turn

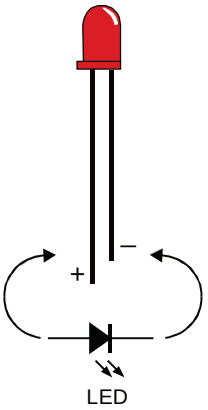
- ✓ Use the table and picture above to figure out the color code for the 220 Ω resistors you will need for the indicator lights.

Introducing the LED

A *diode* is a one-way electric current valve, and a *light-emitting diode* (LED) emits light when current passes through it. Since an LED is a one-way current valve, you have to make sure to connect it the right way for it to work as intended.

An LED has two terminals: the *anode* and the *cathode*. The anode lead is labeled with the plus-sign (+) in the part drawing, and it is the wide part of the triangle in the schematic symbol. The cathode lead is the pin labeled with a minus-sign (-), and it is the line across the point of the triangle in the schematic symbol.

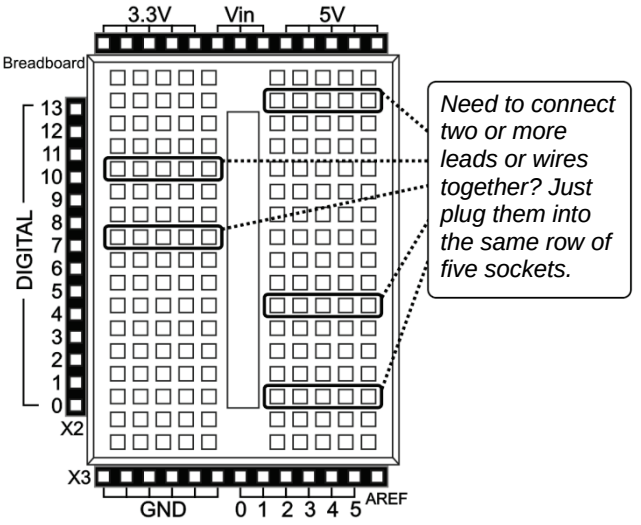
When you build an LED circuit, you will have to make sure the anode and cathode leads are connected to the circuit properly. You can tell them apart by the shape of the LED's plastic case. Look closely at the case—it's mostly round, but there is a small flat spot right near one of the leads, and that tells you it's the cathode. Also note that the LED's leads are different lengths. Usually, the shorter lead is connected to the cathode.



Always check the LED's plastic case. Usually, the longer lead is connected to the LED's anode, and the shorter lead is connected to its cathode. But sometimes the leads have been clipped to the same length, or a manufacturer does not follow this convention. So, it's best to always look for the flat spot on the case. If you plug an LED in backwards, it will not hurt it, but it won't emit light until you plug it in the right way.

Introducing the Prototyping Area

On your Board of Education Shield (and also in the next illustration), the white board with lots of square sockets in it is called a *solderless breadboard*. This breadboard has 17 rows of sockets. In each row, there are two five-socket groups separated by a trench in the middle. All the sockets in a 5-socket group are connected together underneath with a conductive metal clip. So, two wires plugged into the same 5-socket group make electrical contact. This is how you will connect components, such as an LED and resistor, to build circuits. Two wires in the same row on opposite sides of the center trench will not be connected.



The prototyping area also has black sockets along the top, bottom, and left.

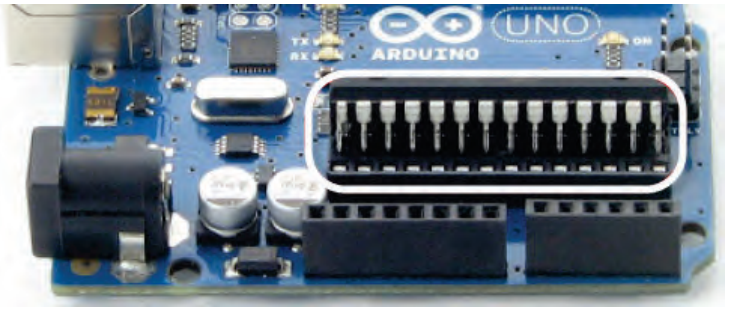
Top: these sockets have three supply voltages for the breadboard: 3.3 V, Vin (input voltage from either battery pack or programming cable), and 5 V.

Bottom-left: The first six sockets along the bottom-left are ground terminals, labeled GND; think of them as a supply voltage that's 0 V. Collectively, the 3.3V, Vin (voltage-in), 5V and GND are called the *power terminals*, and they will be used to supply your circuits with electricity.

Bottom-right: The ANALOG IN sockets along the bottom-right are for measuring variable voltages; these connect to the Arduino module's ANALOG IN sockets.

Left: The DIGITAL sockets on the left have labels from 0 to 13. You will use these to connect your circuit to the Arduino module's digital input/output pins.

Digital and analog pins are the small pins on the Arduino module's Atmel microcontroller chip. These pins electrically connect the microcontroller brain to the board.



A sketch can make the digital pins send high (5 V) or low (0 V) signals *to* circuits. In this chapter, we'll do that to turn lights on and off. A sketch can also make a digital pin monitor high or low signals coming *from* a circuit; we'll do that in another chapter to detect whether a contact switch has been pressed or released.

A sketch can also measure the voltages applied to analog pins; we'll do that to measure light with a phototransistor circuit in another chapter.

LED Test Circuit

Parts List

(2) LEDs – Red
(2) Resistors, 220 Ω (red-red-brown)
(3) Jumper wires

Always disconnect power to your board before building or modifying circuits!

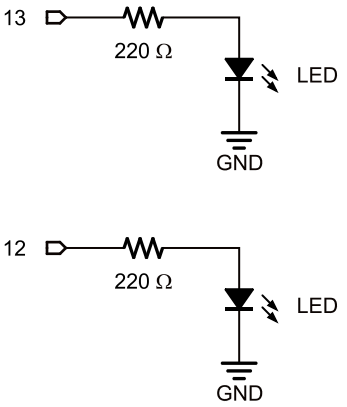
1. Set the BOE Shield's Power switch to 0.
2. Disconnect the programming cable and battery pack.

Building the LED Test Circuits

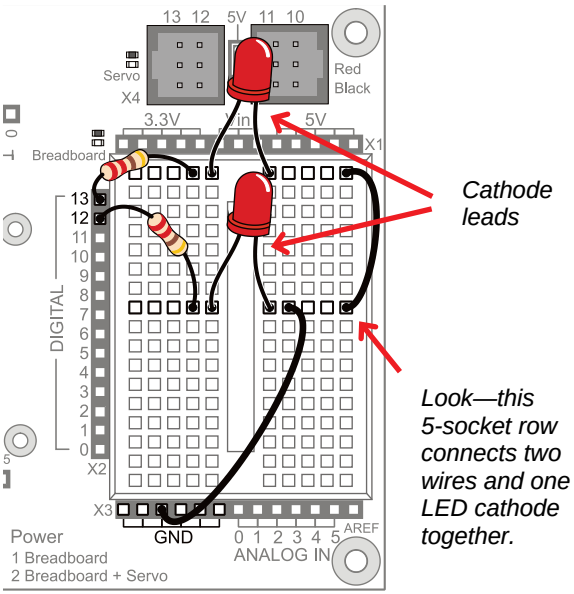
The image below shows the indicator LED circuit schematic on the left, and a wiring diagram example of the circuit built on your BOE Shield's prototyping area on the right.

- ✓ Build the circuit shown below. If you are new to building circuits, try to follow the wiring diagram exactly.
- ✓ Make sure your LED cathode leads are connected to GND. Remember, the cathode leads are the shorter pins that are closer to the flat spot on the LED's plastic case. Each cathode lead should be plugged into the same 5-socket row as the wires that run to the GND sockets.
- ✓ Make sure that each longer anode lead is connected to the same 5-socket row as a resistor lead.

Schematic

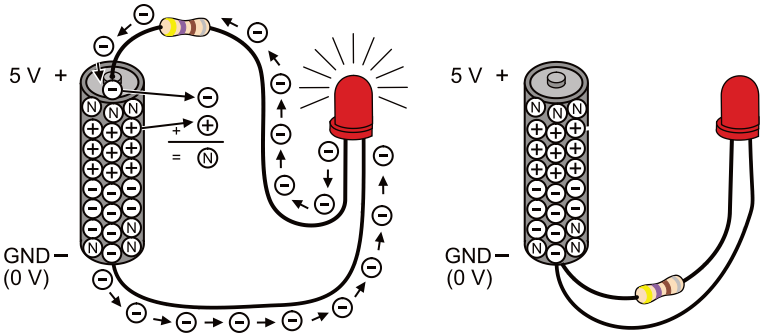


Wiring Diagram



The next picture will give you an idea of what is going on when you program the Arduino to control the LED circuit. Imagine that you have a 5 volt (5 V) battery. The Board of Education Shield has a device called a *voltage regulator* that supplies 5 volts to the sockets labeled 5V. When you connect the anode end of the LED circuit to 5 V, it's like connecting it to the

positive terminal of a 5 V battery. When you connect the circuit to GND, it's like connecting to the negative terminal of the 5 V battery.



On the left side of the picture, one LED lead is connected to 5 V and the other to GND. So, 5 V of electrical pressure causes electrons to flow through the circuit (electric current), and that current causes the LED to emit light. The circuit on the right side has both ends of the LED circuit connected to GND. This makes the voltage the same (0 V) at both ends of the circuit. No electrical pressure = no current = no light.

You can connect the LED to a digital I/O pin and program the Arduino to alternate the pin's output voltage between 5 V and GND. This will turn the LED light on/off, and that's what we'll do next.

Volts is abbreviated V. When you apply voltage to a circuit, it's like applying electrical pressure. By convention, 5 V means "5 V higher than ground." Ground, often abbreviated GND, is considered 0 V.

Ground is abbreviated GND. The term ground originated with electrical systems where this connection is actually a metal rod that has been driven into the ground. In portable electronic devices, ground is commonly used to refer to connections that go to the battery supply's negative terminal.

Current refers to the rate at which electrons pass through a circuit. You will often see measurements of current expressed in amps, which is abbreviated A. The currents you will use here are measured in thousandths of an amp, or milliamps. For example, 10.3 mA passes through the circuit shown previously.

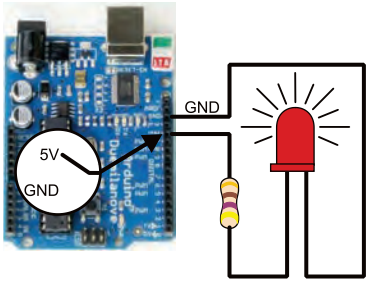
How a Sketch Makes the LED Turn On and Off

Let's start with a sketch that makes the LED circuit connected to digital pin 13 turn on/off. First, your sketch has to tell the Arduino to set the direction of pin 13 to output, using the **`pinMode`** function: **`pinMode(pin, mode)`**. The ***`pin`*** parameter is the number of a digital I/O pin, and ***`mode`*** must be either **`INPUT`** or **`OUTPUT`**.

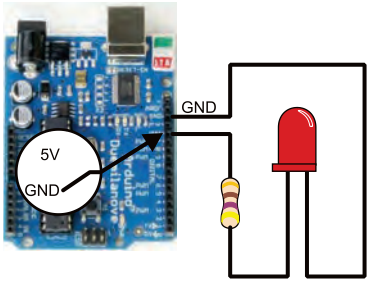
```
void setup()                      // Built-in initialization block
{
  pinMode(13, OUTPUT);            // Set digital pin 13 -> output
}
```

Now that digital pin 13 is set to output, we can use **digitalWrite** to turn the LED light on and off. Take a look at the next picture. On the left, **digitalWrite(13, HIGH)** makes the Arduino's microcontroller connect digital pin 13 to 5 V, which turns on the LED. On the right, it shows how **digitalWrite(13, LOW)** makes it connect pin 13 to GND (0 V) to turn the LED off.

```
digitalWrite(13, HIGH);
```




```
digitalWrite(13, LOW);
```



Here's the **loop** function from the next sketch. First, **digitalWrite(13, HIGH)** turns the light on, **delay(500)** keeps it on for a half-second. Then **digitalWrite(13, LOW)** turns it off, and that's also followed by **delay(500)**. Since it's inside the **loop** function's block, the statements will repeat automatically. The result? The light will flash on/off once every second.

```
void loop()                       // Main loop auto-repeats
{
  digitalWrite(13, HIGH);         // Pin 13 = 5 V, LED emits light
  delay(500);                     // ..for 0.5 seconds
  digitalWrite(13, LOW);          // Pin 13 = 0 V, LED no light
  delay(500);                     // ..for 0.5 seconds
}
```

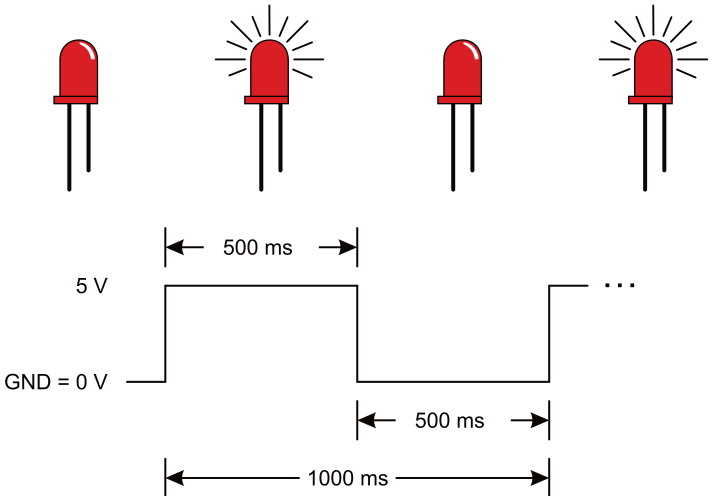
Example Sketch: HighLowLed

- ✓ Reconnect the programming cable to your board.
- ✓ Create and save the HighLowLed sketch, and run it on your Arduino.
- ✓ Verify that the pin 13 LED turns on and off, once every second. (You may see the LED flicker a few times before it settles down into a steady blinking pattern. This happens when reprogramming the Arduino.)

```
/*
 Robotics with the BOE Shield - HighLowLed
 Turn LED connected to digital pin 13 on/off once every second.
 */

void setup()                                    // Built-in initialization block
{
  pinMode(13, OUTPUT);                          // Set digital pin 13 -> output
}

void loop()                                     // Main loop auto-repeats
{
  digitalWrite(13, HIGH);                       // Pin 13 = 5 V, LED emits light
  delay(500);                                   // ..for 0.5 seconds
  digitalWrite(13, LOW);                        // Pin 13 = 0 V, LED no light
  delay(500);                                   // ..for 0.5 seconds
}
```

Introducing the Timing Diagram

A *timing diagram* is a graph that relates a signal's high and low stages to time. This timing diagram shows you a 1000 ms slice of the **`HIGH`** (5 V) and **`LOW`** (0 V) signals from the sketch HighLowLed. Can you see how **`delay(500)`** is controlling the blink rate?



Your Turn – Experiment with the Blink Rates and Both LEDs

How would you make the LED blink twice as fast? How about reducing the **`delay`** function's ms parameters by half?

- ✓ Try modifying your sketch to use **`delay(250)`**. Don't forget to change it in both places!
- ✓ How far can you reduce the delay before it just looks like the LED is dim instead of blinking on/off? Try it!

Blinking the pin 12 LED is a simple matter of changing the ***`pin`*** parameter in the **`pinMode`** and two **`digitalWrite`** function calls.

- ✓ Modify the sketch so that **`pinMode`** in the **`setup`** function uses pin 12 instead of pin 13.
- ✓ Also modify both **`digitalWrite`** statements in the **`loop`** function to use pin 12.
- ✓ Run it, and make sure the pin 12 LED blinks.
- ✓ Make both LEDs blink at the same time, by adding statements to the sketch using **`pinMode`** twice:

```
pinMode(13, OUTPUT);          // Set digital pin 13 -> output
pinMode(12, OUTPUT);          // Set digital pin 12 -> output
```

….and using **`digitalWrite`** four times:

```
digitalWrite(13, HIGH);       // Pin 13 = 5 V, LED emits light
digitalWrite(12, HIGH);       // Pin 12 = 5 V, LED emits light
delay(500);                   // ..for 0.5 seconds
digitalWrite(13, LOW);        // Pin 13 = 0 V, LED no light
digitalWrite(12, LOW);        // Pin 12 = 0 V, LED no light
delay(500);                   // ..for 0.5 seconds
```

- ✓ Run the modified sketch. Do both LEDs blink on and off together?

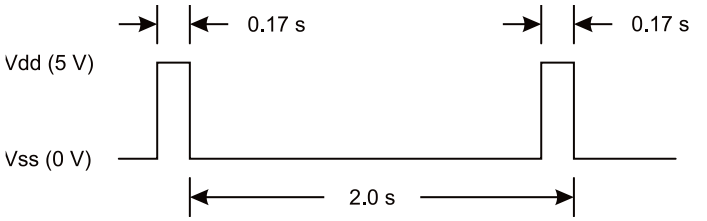
How would you modify the sketch again to turn one LED on while the other turns off? One circuit will need to receive a **`HIGH`** signal while the other receives a **`LOW`** signal.

- ✓ Try it!

Activity 3: LED Servo Signal Monitors

The high and low signals that control servo motors must last for very precise periods of time. That's because a servo motor measures how long the signal stays high, and uses that as an instruction for how fast, and in which direction, to turn its motor.

This timing diagram shows a servo signal that would make your Shield-Bot's wheel turn full speed counterclockwise. There's one big difference though: all the signals in this timing diagram last 100 times longer than they would if they were controlling a servo. This slows it down enough so that we can see what's going on.



Example Sketch: ServoSlowMoCcw

- ✓ Create and save ServoSlowMoCcw, then run it on the Arduino.
- ✓ Verify that the pin 13 LED circuit pulses briefly every two seconds.

```
/*
    Robotics with the BOE Shield - ServoSlowMoCcw
    Send 1/100th speed servo signals for viewing with an LED.
*/

void setup()                            // Built in initialization block
{
  pinMode(13, OUTPUT);                  // Set digital pin 13 -> output
}

void loop()                             // Main loop auto-repeats
{
  digitalWrite(13, HIGH);               // Pin 13 = 5 V, LED emits light
  delay(170);                           // ..for 0.17 seconds
  digitalWrite(13, LOW);                // Pin 13 = 0 V, LED no light
  delay(1830);                          // ..for 1.83 seconds
}
```

Your Turn – Two Steps to Servo Signal

Alright, how about 1/10th speed instead of 1/100th speed?

- ✓ Reduce **`delay(170)`** to **`delay(17)`**, and **`delay(1830)`** to **`delay(183)`**, and re-load the sketch.
- ✓ Is the LED blinking 10 times faster now? Divide by 10 again for a full speed servo signal—we'll have to round the numbers a bit:
- ✓ Change **`delay(17)`** to **`delay(2)`**, and **`delay(183)`** to **`delay(18)`**, then run the modified sketch.

Now you can see what the servo signal looks like with the indicator LED. The LED is flickering so fast, it's just a glow. Since the high signal is 2 ms instead of 1.7 ms, it'll be a little brighter than the actual servo control signal—the light is spending more time on. We could use this signal and programming technique to control a servo, but there's an easier, more precise way. Let's try it with LEDs first.

How to Use the Arduino Servo Library

A better way to generate servo control signals is to include the Arduino Servo library in your sketch, one of the standard libraries of pre-written code bundled with the Arduino software.

To see a list of Arduino libraries, click the Arduino software's Help menu and select Reference. Or, if using codebender : edu, go to the Arduino Library reference page: https://www.arduino.cc/en/Reference/Libraries

- ✓ Find and follow the Libraries link.
- ✓ Find and follow the link to the Servo library.
- ✓ Follow and read the links for these functions on the Servo library page:
  `attach()`
  `writeMicroseconds()`
  `detach()`

Servos have to receive high-pulse control signals at regular intervals to keep turning. If the signal stops, so does the servo. Once your sketch uses the Servo library to set up the signal, it can move on to other code, like delays, checking sensors, etc. Meanwhile, the servo keeps turning because the Servo library keeps running in the background. It regularly interrupts the execution of other code to initiate those high pulses, doing it so quickly that it's practically unnoticeable.

Using the Servo library to send servo control signals takes four steps:

1. Tell the Arduino editor that you want access to the Servo library functions with the #include declaration at the start of your sketch, before the **`setup`** function.

```
#include <Servo.h>          // Include servo library
```

2. Declare and name an instance of the Servo library for each signal you want to send, between **`#include`** and the **`setup`** function.

```
Servo servoLeft;            // Declare left servo
```

3. In the **`setup`** function, use the name you gave the servo signal followed by a dot, and then the attach function call to attach the signal pin. This example is telling the system that the servo signal named **`servoLeft`** should be transmitted by digital pin 13.

```
servoLeft.attach(13);      // Attach left signal to pin 13
```

4. Use the **`writeMicroseconds`** function to set the pulse time. You can do this inside either the **`setup`** or **`loop`** function:

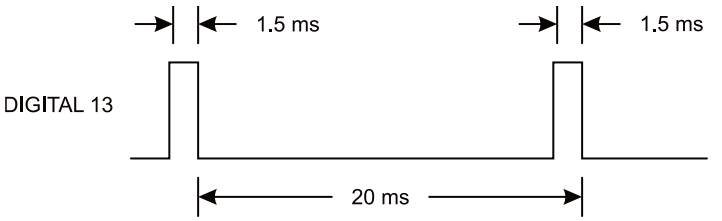
```
servoLeft.writeMicroseconds(1500); // 1.5 ms stay-still signal
```

Seconds, Milliseconds, Microseconds

- A **millisecond** is a one-thousandth of a second, abbreviated ms.
- A **microsecond** is a one-millionth of a second, abbreviated µs.
- There are 1000 microseconds (µs) in 1 millisecond (ms).
- There are 1,000,000 microseconds in 1 second (s).

Example Sketch: LeftServoStayStill

For calibrating servos, your sketch will need to send signals with 1.5 ms pulses. Take a look at the timing diagram below. This stay-still signal's high pulses last 1.5 ms. That's halfway between the 1.7 ms full-speed-counterclockwise and 1.3 ms full-speed-clockwise pulses.



✓ Create and save the LeftServoStayStill sketch, and run it on your Arduino. The pin 13 LED should glow, about halfway between the two brightness levels you observed earlier.

```
/*
 Robotics with the BOE Shield – LeftServoStayStill
 Generate signal to make the servo stay still for centering.
 */

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left servo

void setup()                            // Built in initialization block
{
  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoLeft.writeMicroseconds(1500);    // 1.5 ms stay still signal
}
```

```
void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
}
```

Your Turn – Check a Second Control Signal with the Pin 12 LED

You'll be using this code a lot, so it's a good idea to practice declaring an instance of Servo, attaching the signal to a pin, and setting the pulse duration.

- ✓ Save LeftServoStayStill as BothServosStayStill.
- ✓ Add a second Servo declaration and name it **`servoRight`**.

```
Servo servoRight;             // Declare right servo
```

- ✓ Attach your **`servoRight`** signal to digital pin 12.

```
servoRight.attach(12);        // Attach right signal to pin 12
```

- ✓ Set the **`servoRight`** signal for 1.5 ms (1500 µs) pulses.

```
servoRight.writeMicroseconds(1500); // 1.5 ms stay still signal
```

Your sketch should now look like BothServosStayStill.

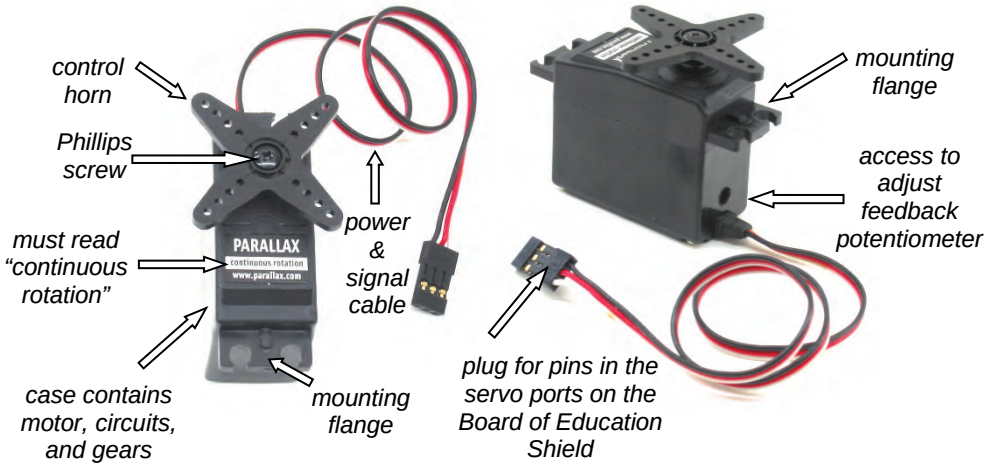
- ✓ Save the sketch and run it on your Arduino.
- ✓ Verify that both LEDs are at a similar brightness level.

```
/*
 Robotics with the BOE Shield – BothServosStayStill
 Generate signals to make the servos stay still for centering.
 */

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left servo signal
Servo servoRight;                       // Declare right servo signal

void setup()                            // Built in initialization block
{
  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach left signal to pin 12

  servoLeft.writeMicroseconds(1500);    // 1.5 ms stay still sig, pin 13
  servoRight.writeMicroseconds(1500);   // 1.5 ms stay still sig, pin 12
}

void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
```

```
}
```

Activity 4: Connect Servo Motors and Batteries

From the robot navigation standpoint, continuous rotation servos offer a great combination of simplicity, usefulness and low price. The Parallax continuous rotation servos are the motors that will make the BOE Shield-Bot's wheels turn, under Arduino control.



In this activity, you will connect your servos to the Board of Education Shield's servo ports, which will connect them to supply voltage, ground, and a signal pin. You will also connect a battery supply to your Arduino because, under certain conditions, servos can end up demanding more current than a USB supply is designed to deliver.

Servo Control Horn, 4-point Star vs. Round
It doesn't make a difference. So long as it is labeled "continuous rotation" it's the servo for your BOE Shield-Bot. You'll remove the control horn and replace it with a wheel.

Standard Servos vs. Continuous Rotation Servos
Standard servos are designed to receive electronic signals that tell them what position to hold. These servos control the positions of radio controlled airplane flaps, boat rudders, and car steering. Continuous rotation servos receive the same electronic signals, but instead turn at certain speeds and directions. Continuous rotation servos are handy for controlling wheels and pulleys.

Connect the Servos to the BOE Shield

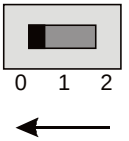
Leave the LED circuits from the last activity on your board. They will be used later to monitor the signals the Arduino sends to the servos to control their motion.

Parts List

(2) Parallax continuous rotation servos
(1) BOE Shield with built and tested LED indicator circuits from the previous activity

Instructions

✓ Set your Shield's power switch to position-0.

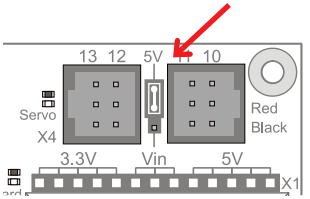


✓ Disconnect all sources of power from the Arduino **including the USB cable**.

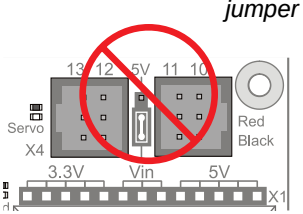
Between the servo headers on the BOE Shield is a jumper that connects the servo power supply to either Vin or 5V. To move it, pull it upwards and off the pair of pins it covers, then push it onto the pair of pins you want it to rest on. The BOE Shield-Bot's battery pack will supply 7.5 V. Since the servos are rated for 4–6 V, we want to make sure the jumper is set to 5V. Also, a steady 5 V voltage supply will support a consistent servo speed, and more accurate navigation, than voltage that varies as batteries discharge.

✓ Make sure your BOE Shield's power jumper is set to 5V; if not, set it now.

This jumper is set to 5 V.

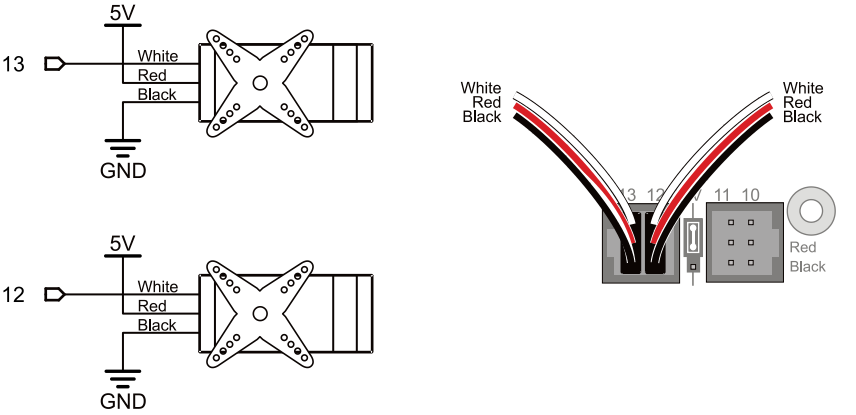


DO NOT set the jumper to Vin!



The picture below shows the schematic of the circuit you create by plugging the servos into ports 13 and 12 on the BOE Shield. Pay careful attention to wire color as you plug in the cables: the black wire should be at the bottom, and the white one should be at the top.

- ✓ Connect your servos to your BOE Shield as shown in the diagram below. The left servo connects to port 13 and the right servo connects to port 12.
- ✓ Make sure that you follow the cable colors shown in the figure, with the black wire closer to the breadboard and the white wire closer to the board's edge.



Connect the Battery Pack to the BOE Shield

To properly power the servos, you'll need to switch to an external battery pack now. When servos make sudden direction changes or push against resistance to rotation, they can draw more current than a USB port is designed to supply. Also, it would be no fun for the BOE Shield-Bot to be tethered to the computer forever! So, from here on out we'll be using an external battery pack with five 1.5 V AA batteries. This will supply your system with 7.5 V and plenty of current for the voltage regulators and servos. From here forward, remember two things:

1. **ALWAYS unplug the battery pack** when you are done experimenting for a while. Even when the power switch on your BOE Shield is off (position-0), the Arduino module will still draw power from the batteries.
2. **Unplug the programming cable too**, whenever you unplug the battery pack. That way, you won't accidentally try to run the servos off of USB power.

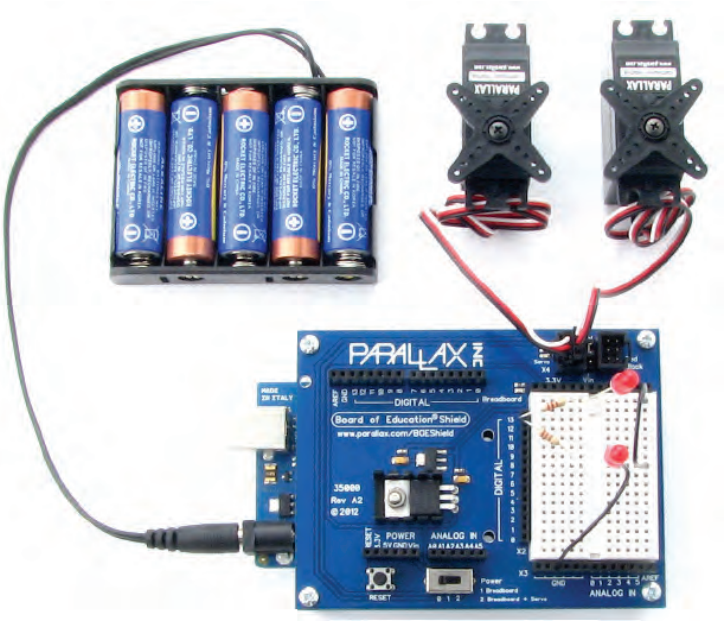
CAUTION: AC powered DC supplies are not recommended for the BOE Shield-Bot.
Some DC supplies provide much higher voltage than their rating. The BOE Shield-bot is designed for use with a 7.2–7.5 V battery supply. It will work with higher supply voltages at low loads, but the servo loads can heat up the regulator until it shuts off to protect itself.

Parts List

(5) 1.5 volt AA batteries
(1) 5-cell battery pack

Load the Batteries

- ✓ Load the batteries into the battery pack.
- ✓ Plug the battery pack into the Arduino's power jack. When you are done, it should resemble the picture below.



Activity 5: Centering the Servos

In this activity, you will run a sketch that sends the "stay-still" signal to the servos. You will then use a screwdriver to adjust the servos so that they actually stay still. This is called *centering* the servos. After the adjustment, you will run test sketches that will turn the servos clockwise and counterclockwise at various speeds.

Tool Required

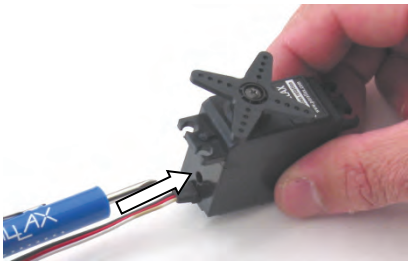
You'll need a Phillips #1 point screwdriver with a 1/8″ (3.18 mm) or smaller shaft.

Sending the Center Signals

If a servo has not yet been centered, it may turn, vibrate, or make a humming noise when it receives the "stay-still" signal.

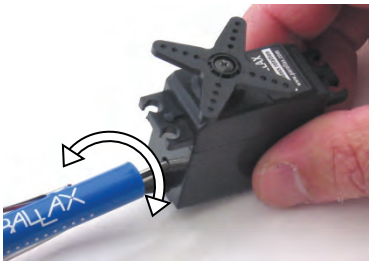
- ✓ Reconnect your programming cable, and re-load Example Sketch: LeftServoStayStill from page 58.
- ✓ Set the BOE Shield's Power switch to 2, to provide power to the servos.

- ✓ Use a screwdriver to gently adjust the potentiometer in the servo as shown below. Don't push too hard! Adjust the potentiometer slightly until you find the setting that makes the servo stop turning, humming or vibrating.



Insert tip of screwdriver into potentiometer access hole,

…then twist potentiometer until the servo is completely still.



- ✓ Verify that the pin 13 LED signal monitor circuit is showing activity. It should glow like it did when you ran LeftServoStayStill the first time.

What's a Potentiometer? It is kind of like an adjustable resistor with a moving part, such as a knob or a sliding bar, for setting the resistance. The Parallax continuous rotation servo's potentiometer is a recessed knob that can be adjusted with a small Phillips screwdriver tip.

Your Turn – Center the Servo Connected to Pin 12

✓ Repeat the process for the pin 12 servo using the sketch RightServoStayStill.

```
/*
 Robotics with the BOE Shield – RightServoStayStill
 Transmit the center or stay still signal on pin 12 for center adjustment.
 */

#include <Servo.h>                      // Include servo library

Servo servoRight;                       // Declare right servo

void setup()                            // Built-in initialization block
{
  servoRight.attach(12);                // Attach right signal to pin 12
  servoRight.writeMicroseconds(1500);   // 1.5 ms stay still signal
}

void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
}
```

Activity 6: Testing the Servos

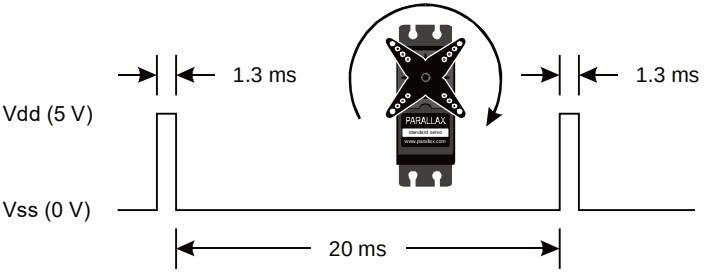
There's one last thing to do before assembling your BOE Shield-Bot, and that's testing the servos. In this activity, you will run sketches that make the servos turn at different speeds and directions. This is an example of *subsystem testing*—a good habit to develop.

Subsystem testing is the practice of testing the individual components before they go into the larger device. It's a valuable strategy that can help you win robotics contests.

Engineers use this essential skill to develop everything from toys, cars, and video games to space shuttles and Mars roving robots. Especially in more complex devices, it can become nearly impossible to figure out a problem if the individual components haven't been tested beforehand. In aerospace projects, for example, disassembling a prototype to fix a problem can cost hundreds of thousands, or even millions, of dollars. In those kinds of projects, subsystem testing is rigorous and thorough.

Pulse Width Controls Speed and Direction

This timing diagram shows how a Parallax continuous rotation servo turns full speed clockwise when you send it 1.3 ms pulses. Full speed falls in the 50 to 60 RPM range.



What's RPM? Revolutions Per Minute—the number of full rotations turned in one minute.
What's a pulse train? Just as a railroad train is a series of cars, a pulse train is a series of pulses (brief high signals).

Example Sketch: LeftServoClockwise

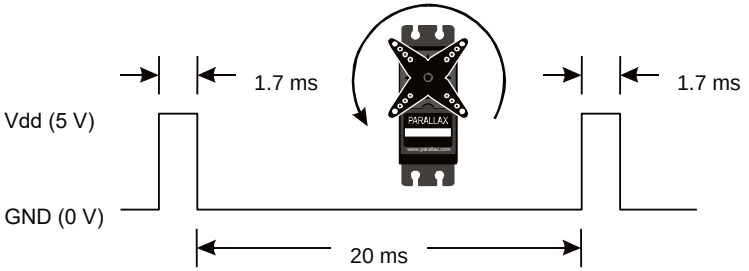
- ✓ Create and save the LeftServoClockwise sketch, and run it on your Arduino.
- ✓ Verify that the servo's horn is rotating between 50 and 60 RPM clockwise.

```
/*
 Robotics with the BOE Shield – LeftServoClockwise
 Generate a servo full speed clockwise signal on digital pin 13.
 */

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left servo

void setup()                            // Built in initialization block
{
  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoLeft.writeMicroseconds(1300);    // 1.3 ms full speed clockwise
}

void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
}
```

Your Turn: Left Servo Counterclockwise

- ✓ Save LeftServoClockwise as LeftServoCounterclockwise.
- ✓ In **`servoLeft.writeMicroseconds`**, change **`(1300)`** to **`(1700)`**.

- ✓ Save the modified sketch and run it on the Arduino.
- ✓ Verify that the servo connected to pin 13 now rotates the other direction, which should be counterclockwise, at about 50 to 60 RPM.



Example Sketch: RightServoClockwise

- ✓ Save LeftServoClockwise as RightServoClockwise.
- ✓ Replace all instances of **servoLeft** with **servoRight**.
- ✓ Replace all instances of **13** with **12**.
- ✓ Save and run the modified sketch and verify that the pin 12 servo is rotating between 50 and 60 RPM clockwise.

```
/*
 Robotics with the BOE Shield – RightServoClockwise
 Generate a servo full speed clockwise signal on digital pin 12.
 */

#include <Servo.h>                      // Include servo library

Servo servoRight;                       // Declare left servo

void setup()                            // Built in initialization block
{
  servoRight.attach(12);                // Attach left signal to pin 12
  servoRight.writeMicroseconds(1300);   // 1.3 ms full speed clockwise
}

void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
}
```

Your Turn – Right Servo Counterclockwise

- ✓ In **servoRight.writeMicroseconds** change **(1300)** to **(1700)**.
- ✓ Save the sketch under a new name and run it on your Arduino.
- ✓ Verify that the pin 12 servo turns full-speed counterclockwise, 50–60 RPM.

Controlling Servo Speed and Direction

For BOE Shield-Bot navigation, we need to control both servos at once.

- ✓ Create and save ServosOppositeDirections, and run it on the Arduino.
- ✓ Verify that the servo connected to pin 13 turns counterclockwise and the one connected to pin 12 turns clockwise.

Example Sketch: ServosOppositeDirections

```
 /*
 Robotics with the BOE Shield – ServosOppositeDirections
 Generate a servo full speed counterclockwise signal with pin 13 and
 full speed clockwise signal with pin 12.
 */

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left servo signal
Servo servoRight;                       // Declare right servo signal

void setup()                            // Built in initialization block
{
  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12

  servoLeft.writeMicroseconds(1700);    // 1.7 ms -> counterclockwise
  servoRight.writeMicroseconds(1300);   // 1.3 ms -> clockwise
}

void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
}
```

This opposite-direction control will be important soon. Think about it: when the servos are mounted on either side of a chassis, one will have to rotate clockwise while the other rotates counterclockwise to make the BOE Shield-Bot roll in a straight line. Does that seem odd? If you can't picture it, try this:

- ✓ Hold your servos together back-to-back while the sketch is running.

Different combinations of **`writeMicroseconds`** ***`us`*** parameters will be used repeatedly for programming your BOE Shield-Bot's motion. By testing several possible combinations and filling in the Description column of the table below, you will become familiar with them and build a reference for yourself. You'll fill in the Behavior column later on, when you see how the combinations make your assembled BOE Shield-Bot move.

Pulse Width Modulation: Adjusting the property of a signal to carry information is called *modulation*. We've discovered that servo control signals are a series of high pulses separated by low resting states. How long the high pulse lasts—how wide the high pulse looks in a timing diagram—determines the speed and direction that the servo turns. That adjustable pulse width carries the servo setting information. Therefore, we can say that servos are controlled with *pulse width modulation*, abbreviated PWM.

- ✓ Save a copy of the ServosOppositeDirections sketch as TestServosTable2_2.
- ✓ Test each combination of values in **`servoLeft.WriteMicroseconds(us)`** and **`servoRight.writeMicroseconds(us)`** and record your results in the table.

writeMicroseconds(us) Combinations			
Pin 13 servoLeft	**Pin 12 servoRight**	**Description**	**Behavior**
1700	1300	*Full speed, pin 13 servo counter-clockwise, pin 12 servo clockwise.*	
1300	1700		
1700	1700		
1300	1300		
1500	1700		
1300	1500		
1500	1500	*Both servos should stay still.*	*Robot stays still.*
1520	1480		
1540	1460		
1700	1450		
1550	1300		

How To Control Servo Run Time

It's easy to control how long the servos run when using the Servo library. Once set, a servo will maintain its motion until it receives a new setting. So, to make a servo run for a certain length of time, all you have to do is insert a **`delay`** after each setting.

Example Sketch: ServoRunTimes

- ✓ Create and save the ServoRunTimes sketch, and run it on your Arduino.
- ✓ Verify that both servos turn full speed clockwise for 3 seconds, then counterclockwise for 3 seconds, then stop.

```
/*
 Robotics with the BOE Shield – ServoRunTimes
 Generate a servo full speed counterclockwise signal with pin 13 and
 full speed clockwise signal with pin 12.
 */

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left servo signal
Servo servoRight;                       // Declare right servo signal

void setup()                            // Built in initialization block
{
  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12

  servoLeft.writeMicroseconds(1300);    // Pin 13 clockwise
  servoRight.writeMicroseconds(1300);   // Pin 12 clockwise
  delay(3000);                          // ..for 3 seconds
  servoLeft.writeMicroseconds(1700);    // Pin 13 counterclockwise
  servoRight.writeMicroseconds(1700);   // Pin 12 counterclockwise
  delay(3000);                          // ..for 3 seconds
  servoLeft.writeMicroseconds(1500);    // Pin 13 stay still
  servoRight.writeMicroseconds(1500);   // Pin 12 stay still
}

void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
}
```

Chapter 2 Summary

The focus of this chapter was calibrating and testing the servos, and building indicator lights to monitor the servo signals. In addition to some hardware setup, many concepts related to electronics, programming, and even a few good engineering concepts were introduced along the way.

Hardware Setup

- How to mount the Board of Education Shield on your Arduino module
- How and why to provide an external battery power supply for the system
- How to connect the servos to the Board of Education Shield

Electronics

- What a resistor does, what its schematic symbol looks like, and how to read its value by decoding the color-bands on its case
- What tolerance is, in relation to a resistor's stated value
- What an LED does, what the schematic symbol for this one-way current valve looks like, and how to identify its anode and cathode
- What a solderless breadboard is for, and how it connects electronic devices
- How to build LED indicator light circuits
- The parts of a servo motor, how to connect it to the Board of Education Shield
- What a potentiometer is, and how to calibrate a Parallax continuous rotation servo by adjusting its potentiometer
- What a pulse train is, and how to control a servo with pulse width modulation

Programming

- How to use the Arduino's **`pinMode`** and **`digitalWrite`** functions to send high and low output signals
- How to use **`#include`** to add the Servo library to a sketch
- Using the Servo library's functions to attach and control a servo
- Writing sketches to control the servo's speed, direction, and run time

Engineering

- Building a circuit on a solderless prototyping area
- Using an indicator light to monitor communication between devices
- What subsystem testing is, and why it is important
- Creating a reference table of input parameters and how they affect a device's output

Chapter 2 Challenges

Questions

1. How do you connect two leads together using a breadboard?
2. What function sets a digital pin's direction?
3. What function sets pin 13 to 5 V? What function sets it to 0 V?
4. How can a sketch control the duration of a 5 V signal?
5. What are the pulse durations that tell a continuous rotation servo to turn
    a. full speed clockwise,
    b. full speed counterclockwise,
    c. stay still.
6. Which call would make a servo turn faster?
    a. **`servoLeft.writeMicroseconds(1440)`** or
    b. **`servoLeft.writeMicroseconds(1420)`**. Why?
7. How can a sketch control the duration of a certain servo signal?

Exercises

1. Write a **`loop`** function that makes an LED blink 5 times per second, with an on time that's 1/3rd of its off time. (Move power switch to position 1 to turn off the servos for this exercise!)
2. Write a **`setup`** function that makes the pin 13 servo turn full speed clockwise for 1.2 seconds, while the pin 12 servo stays still. After that, set both servos to stop.
3. Write a **`setup`** function that makes one servo turn the same direction for 3 seconds. The other servo should turn the opposite direction for the first 1.5 seconds and the same direction for the second 1.5 seconds. Then, make both servos stop.

Projects

1. Look up the servo library's **`detach`** function and use it in place of **`servoLeft`** and **`servoRight.writeMicroseconds(1500)`** to stop servos after they turn for 3 seconds.
2. Write a program that makes the pin 13 servo turn counterclockwise while the pin 12 servo turns clockwise. After 3 seconds, make both servos turn counterclockwise for 0.6 seconds. Next, make both turn clockwise for 0.6 seconds. Then, make the pin 13 servo turn clockwise and the pin 12 servo turn counterclockwise for 3 seconds.

Question Solutions

1. Plug the two leads into the same 5-socket row on the breadboard.
2. The **pinMode** function.
3. The **digitalWrite** function does both, depending on its ***value*** parameter:

```
digitalWrite(13, HIGH) // 5 V
digitalWrite(13, LOW)  // 0 V
```

4. Assuming a pin has just been set high, the **delay** call can keep it high for a certain amount of time. Then, a **digitalWrite** call can set it low.
5. (a)1.3 ms pulses for full speed clockwise, (b)1.7 ms pulses for full speed clockwise, and (c)1.5 ms pulses for stay still.
6. (b) **servoLeft.writeMicroseconds(1420)**. Full speed clockwise is **servoLeft.writeMicroseconds(1300)**, and stop is **servoLeft.WriteMicroseconds(1500)**. Since 1420 is further from stop and closer to full speed, it's the correct value for faster clockwise rotation even though it is smaller.
7. **Servo.writeMicroseconds(*value*)** followed by **delay(*ms*)** followed by **Servo.writeMicroseconds(*newValue*)** or **Servo.detach(*pin*)** will keep the servo turning for ***ms*** milliseconds.

Exercise Solutions

1. The total on + off time has to be 200 ms, which is 1/5th of a second. So, on for 50 ms, off for 150 ms:

```
void loop()                    // Main loop auto-repeats
{                              // 200 ms -> 5 blinks/second
  digitalWrite(13, HIGH);      // Pin 13 = 5 V, LED emits light
  delay(50);                   // ..for 0.05 seconds
  digitalWrite(13, LOW);       // Pin 13 = 0 V, LED no light
  delay(150);                  // ..for 0.15 seconds
}
```

2. Set pin the 13 servo to full speed clockwise and the pin 12 servo to stop. Then, delay for 1200. Since **servoRight** is already stopped, all the code has to do is stop **servoLeft**.

```
void setup()                   // Built in initialization block
{
  servoLeft.attach(13);        // Attach left signal to pin 13
  servoRight.attach(12);       // Attach right signal to pin 12

  servoLeft.writeMicroseconds(1300);  // 1.3 ms -> clockwise
```

```
  servoRight.writeMicroseconds(1500); // 1.5 ms -> stop
  delay(1200);                        // ..for 1.2 seconds
  servoLeft.writeMicroseconds(1500);  // 1.5 ms -> stop
}
```

3. In this example, the pin 13 servo starts counterclockwise and the pin 12 servo starts out clockwise. This goes on for 1.5 seconds. Then, the pin 12 servo is changed to counterclockwise, and this goes on for another 1.5 seconds. After that, both servos are stopped.

```
void setup()                    // Built in initialization block
{
  servoLeft.attach(13);         // Attach left signal to pin 13
  servoRight.attach(12);        // Attach right signal to pin 12

  servoLeft.writeMicroseconds(1700);  // 1.7 ms -> cc-wise
  servoRight.writeMicroseconds(1300); // 1.3 ms -> clockwise
  delay(1500);                        // ..for 1.5 seconds
  servoRight.writeMicroseconds(1700); // 1.7 ms -> cc-wise
  delay(1500);
  servoLeft.writeMicroseconds(1500);  // 1.5 ms -> stop
  servoRight.writeMicroseconds(1500); // 1.5 ms -> stop
}
```

Project Solutions

1. The **`detach`** function detaches the instance of Servo from its pin. This sketch verifies that it stops the servo after 3 seconds of run time. The chapter examples sent pulses telling the servo to stay still. In contrast, **`detach`** stops sending signals to the servo—the pin doesn't tell the servo to do anything, so it goes dormant instead of holding the "stop speed." The end result is the same, the servo motor stops. The advantage to **`detach`** is that it prevents the servos from turning slowly if the servo is not precisely calibrated.

```
/*
 Robotics with the BOE Shield – Chapter 2, Project 1
 Generate a servo full speed counterclockwise signal with pin 13 and
 full speed clockwise signal with pin 12.
 */

#include <Servo.h>                    // Include servo library

Servo servoLeft;                      // Declare left servo signal
Servo servoRight;                     // Declare right servo signal

void setup()                          // Built in initialization block
{
  servoLeft.attach(13);               // Attach left signal to pin 13
```

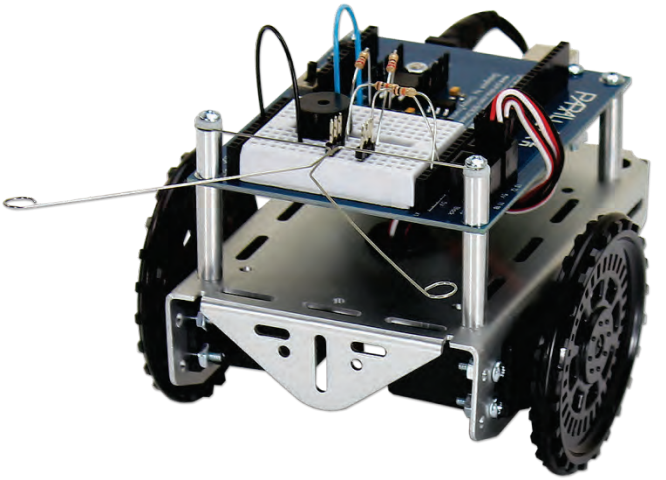
```
  servoRight.attach(12);              // Attach right signal to pin 12

  servoLeft.writeMicroseconds(1700);  // Pin 13 counterclockwise
  servoRight.writeMicroseconds(1300); // Pin 12 clockwise
  delay(3000);                        // ..for 3 seconds
  servoLeft.detach();                 // Stop servo signal to pin 13
  servoRight.detach();                // Stop servo signal to pin 12
}

void loop()                           // Main loop auto-repeats
{                                     // Empty, nothing needs repeating
}
```

2. Solution is the sketch ForwardLeftRightBackward, from a later chapter.

Chapter 3. Assemble and Test your BOE Shield-Bot

This chapter contains instructions for building and testing your BOE Shield-Bot. It's especially important to complete the testing portion before moving on to the next chapter. By doing so, you can help avoid a number of common mistakes that could otherwise lead to mystifying BOE Shield-Bot behavior. Here is a summary of what you will do:

- Build the BOE Shield-Bot.
- Re-test the servos to make sure they are properly connected.
- Connect and test a speaker that can let you know when the BOE Shield-Bot's batteries are running low.
- Use the Serial Monitor to control and test servo speed.



Activity 1: Assembling the BOE Shield-Bot

This activity will guide you through assembling the BOE Shield-Bot, step by step. In each step, you will gather a few of the parts, and then assemble them so that they match the pictures. Each picture has instructions that go with it; make sure to follow them carefully.

Tools

All of the tools needed are common and can be found in most households and school shops. They can also be purchased at local hardware stores. The Parallax screwdriver is included in the Robotics Shield Kit, and the other two are optional but handy to have.

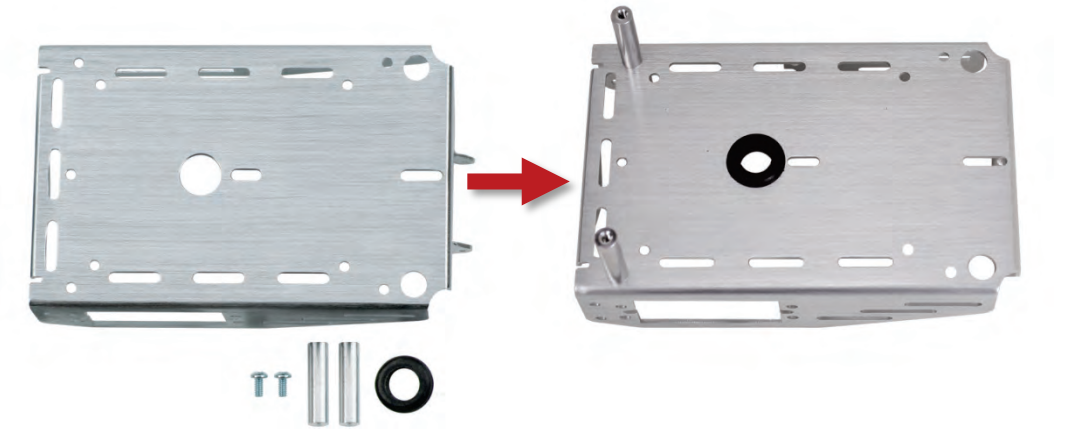
(1) screwdriver, Phillips #1
(1) 1/4″ combination wrench (recommended if you choose to use the Nylon-core lock nuts)
(1) needle-nose pliers (optional)

- ✓ Disconnect the programming cable and battery pack from your Arduino.
- ✓ Disconnect the servos from the BOE Shield.
- ✓ Set the BOE Shield's 3-position power switch to position 0.

Mount the Topside Hardware

Parts List

(1) robot chassis
(2) 1″ standoffs (removed from BOE Shield)
(2) pan-head screws, 1/4″ 4-40 (removed from BOE Shield)
(1) rubber grommet, 13/32″



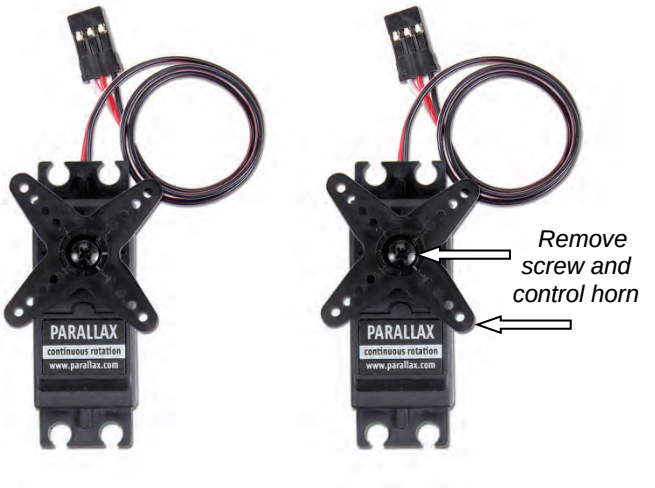
Instructions

- ✓ Remove the 1″ aluminum standoffs from the BOE Shield, and save the standoffs and screws.
- ✓ Insert the 13/32″ rubber grommet into the hole in the center of the chassis.
- ✓ Make sure the groove in the outer edge of the rubber grommet is seated on the metal edge of the hole.
- ✓ Use two of the 1/4″ 4-40 screws to attach two of the standoffs to the top front of the chassis as shown.
- ✓ Save the other two standoffs and screws for a later step.

Remove the Servo Horns

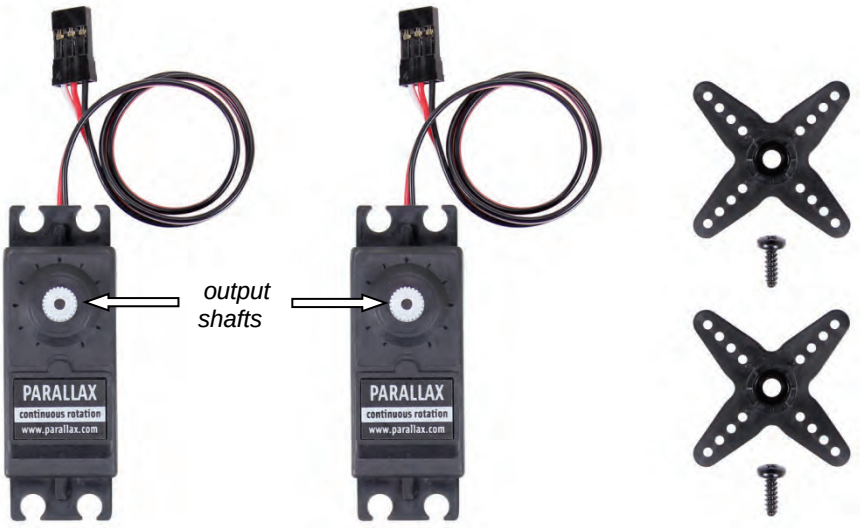
Parts List

(2) Parallax continuous rotation servos, previously centered



Instructions

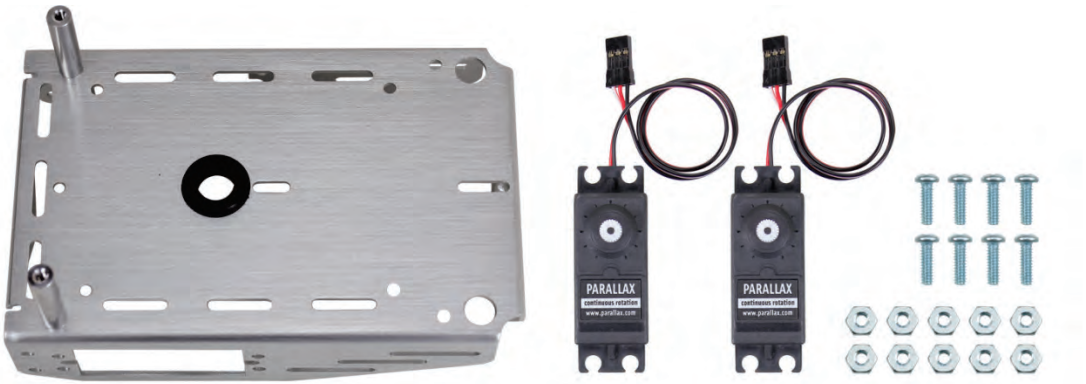
- ✓ Use a Phillips screwdriver to remove the screws that hold the servo control horns on the output shafts.
- ✓ Pull each horn upwards and off the servo output shaft.
- ✓ **Save the screws; you will need them again soon!**



Mount the Servos on the Chassis

Parts List

(2) BOE Shield-Bot Chassis, partially assembled.
(2) Parallax continuous rotation servos
(8) pan Head Screws, 3/8″ 4-40
(8) steel nuts, your choice of regular 4-40 or lock nuts
1/4" combination wrench (optional, needed for lock nuts only)
masking tape & pen

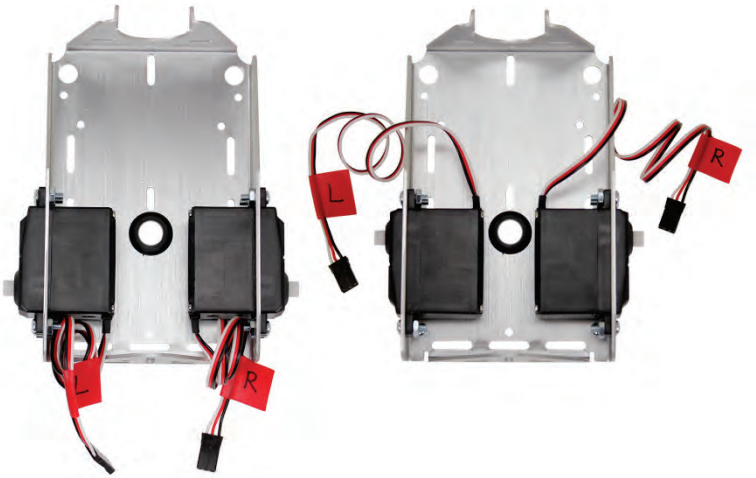


Instructions

STOP! Before taking the next step, you must have completed these Chapter 2 activities: Activity 4: Connect Servo Motors and Batteries on page 60, and Activity 5: Centering the Servos on 63.

Decide how you want to mount your servos from the two options described and pictured below. Option 2 (right) is considered our standard small robot configuration, and our navigation code examples assume you have set up your robot this way. Code examples may need to be adjusted if you choose Option 1 (left).

Nuts Note: You can choose to use either hex nuts or locknuts to mount your servos, both are provided. Locknuts provide a tighter connection, but if your robots need frequent repair or part replacements then hex nuts are the easiest to remove and reattach quickly.



Outside-forward (left) — the servos' mounting tabs seat outside the chassis, with their potentiometer access ports facing toward the front of the chassis. This allows easy access to adjust the potentiometers on an assembled robot, and also makes servo replacement quick. However, this gives the BOE Shield-Bot a longer, wider wheel base, so it will be a little less nimble on maneuvers and may need more pulses to make turns.

Inside-backward (right) — the servos' mounting tabs seat inside the chassis, with their potentiometer access ports facing towards the battery pack. This positions the axles close to the center of the BOE Shield-Bot, for maximum agility. If you are diligent about centering your servos before building your BOE Shield-Bot, this causes no problems.

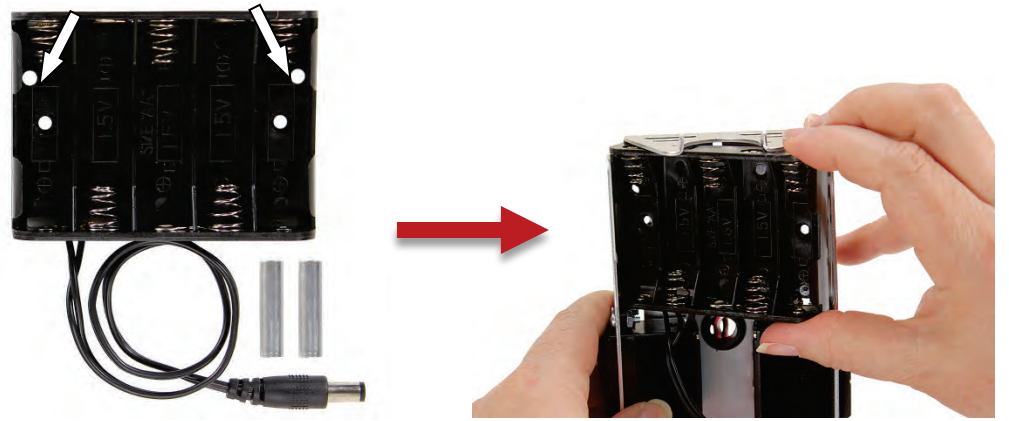
- ✓ Attach the servos to the chassis using the Phillips screws and nuts.
- ✓ Use pieces of masking tape to label the servos left (L) and right (R), as shown.

Mount the Battery Pack

Parts List

(2) Nylon flat-head slotted screws, 3/8″ 4-40
(2) 1" standoffs (removed from BOE Shield previously)
(1) 5-cell battery pack with 2.1 mm center-positive plug

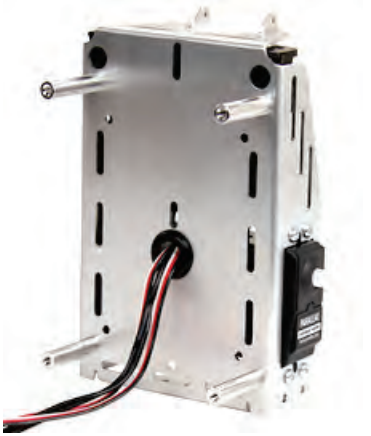
3/8" Nylon screws will go through outer holes and be secured to chassis with 1" standoffs.



Instructions

✓ Place the empty battery pack inside the chassis positioned as shown above. The easiest method is to insert one side, then press down on the other side until it is flush with the chassis. The fit may be snug, but it should snap into place using a small amount of force (below).

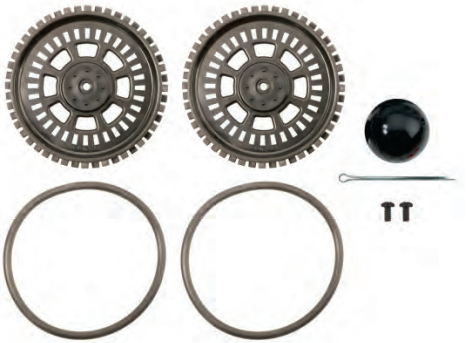
- ✓ Insert the two Nylon flat-head screws through the inside of the battery pack. Use the smaller set of holes that line up with the chassis mounting holes for the front standoffs.
- ✓ From the top of the chassis, thread a 1" standoff on each screw and tighten.
- ✓ Pull the battery pack's power cord and servo lines through the rubber grommet hole in the center of the chassis, as shown below.



Mount the Wheels

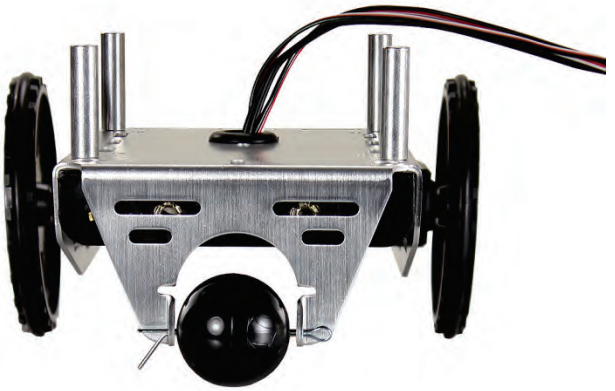
Parts List

(1) 1/16″ cotter pin
(1) tail wheel ball
(2) O-ring tires
(2) plastic machined wheels
(2) screws saved when removing the servo horns



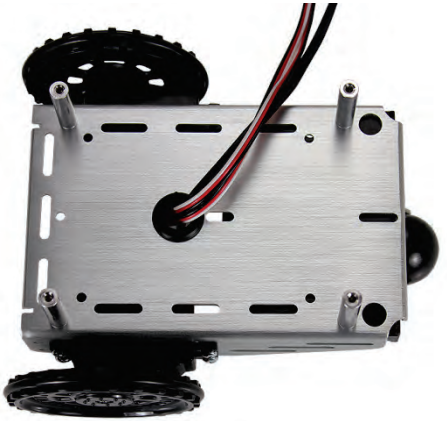
Instructions

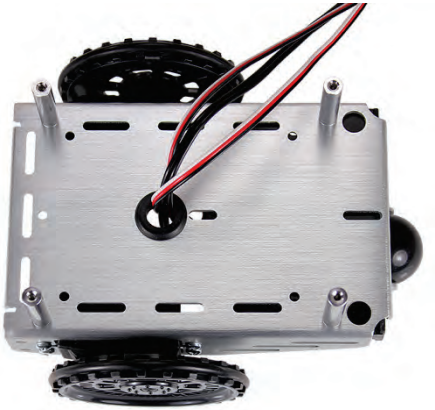
The robot's tail wheel is merely a plastic ball with a hole through the center. A cotter pin holds it to the chassis and functions as an axle for the wheel.



- ✓ Line up the hole in the tail wheel with the holes in the tail portion of the chassis.
- ✓ Run the cotter pin through all three holes (chassis left, tail wheel, chassis right).
- ✓ Bend the ends of the cotter pin apart so that it can't slide back out of the hole.
- ✓ Press each plastic wheel onto a servo output shaft, making sure the shaft lines up with, and sinks into, the wheel's recess, then secure with the saved servo screws.
- ✓ Set an O-ring tire on the outer edge of each wheel, in the channel between the teeth.

When you are done, your completed chassis will look like one of the pictures below. The left shows the "outside-forward" servos, the right shows "inside-backward" servos.





Attach the BOE Shield to the Chassis

Parts List

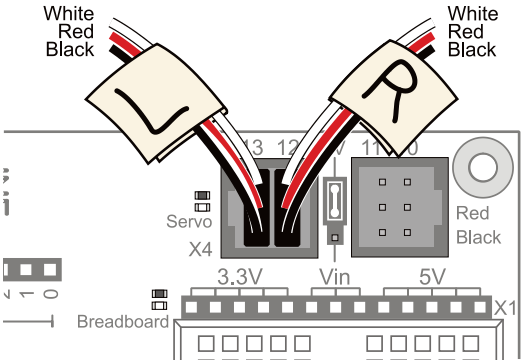
(4) pan-head screws, 1/4″ 4-40
(1) Board of Education Shield mounted to your Arduino module and secured with standoffs.

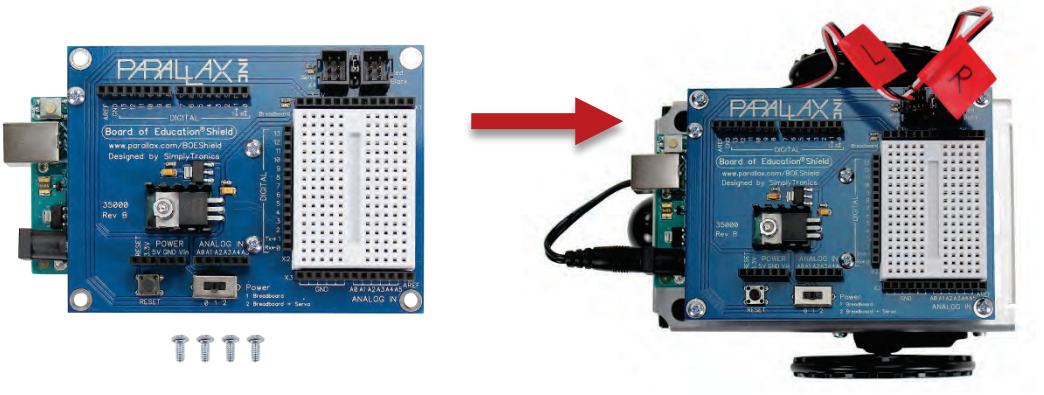
Instructions

- ✓ Set the BOE Shield on the four standoffs, lining them up with the four mounting holes on the outer corner of the board.
- ✓ Make sure the white breadboard is closer to the drive wheels, not the tail wheel.
- ✓ Attach the board to the standoffs with the pan-head screws.
- ✓ Reconnect the servos to the servo headers.

Using Different Pins for the Servos: The Arduino toggles Pin 13 briefly upon startup or reset. If this causes problems for a particular application, you can use Pins 11 and 12 instead of 12 and 13. Be sure to adjust your code accordingly.

ROBOTC: If you are building the BOE Shield-Bot to use with ROBOTC instead, follow the directions here: http://learn.parallax.com/tutorials/setting-boe-shield-bot-servos-robotc

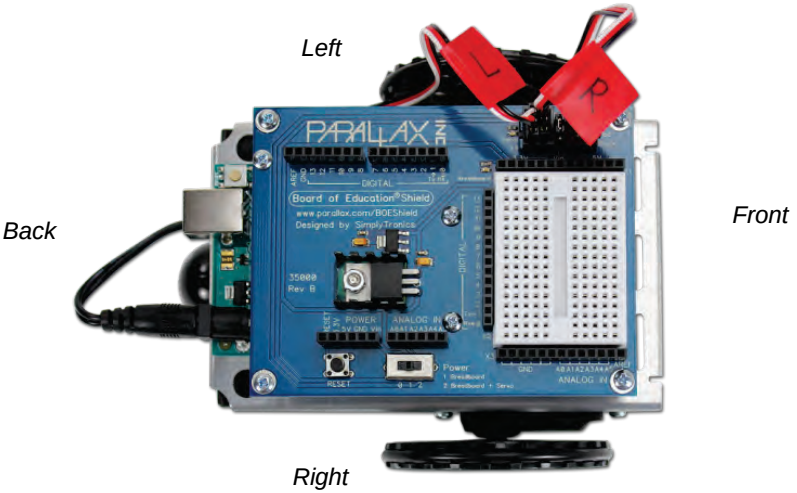




From the underside of the chassis, pull any excess servo and battery cable through the rubber grommet hole, and tuck the excess cable lengths between the servos and the chassis.

Activity 2: Re-Test the Servos

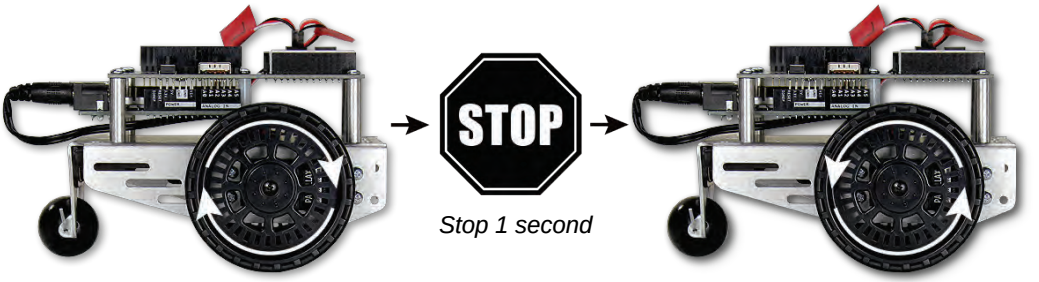
In this activity, you will test to make sure that the electrical connections between your board and the servos are correct. The picture below shows your BOE Shield-Bot's front, back, left, and right. We need to make sure that the right-side servo turns when it receives pulses from pin 12 and that the left-side servo turns when it receives pulses from pin 13.



Testing the Left and Right Wheels

Test the Right Wheel First

The next example sketch will test the servo connected to the right wheel, shown below. The sketch will make this wheel turn clockwise for three seconds, then stop for one second, then turn counterclockwise for three seconds.



Example Sketch: RightServoTest

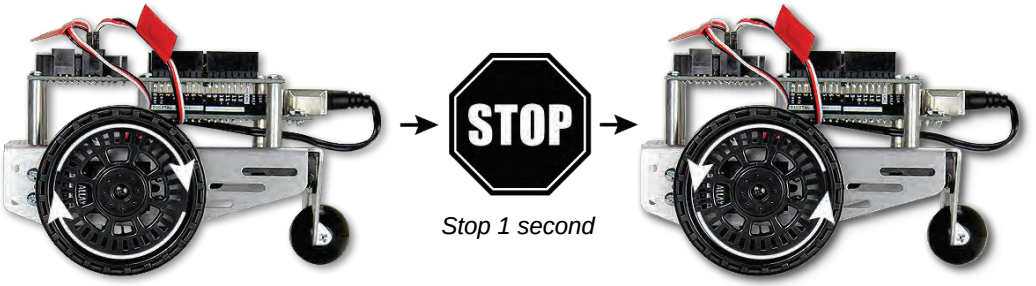
- ✓ Set the BOE Shield-Bot on its nose so that the drive wheels are suspended above the ground.
- ✓ Connect the programming cable and battery pack to the Arduino.
- ✓ Create and save the RightServoTest sketch, and run it on your Arduino.
- ✓ Set the 3-position switch to position-2 and press/release the RESET button.
- ✓ Verify that the right wheel turns clockwise for three seconds, stops for one second, then turns counterclockwise for three seconds.
- ✓ If the right wheel/servo does not behave as predicted, see Servo Troubleshooting, page 88.
- ✓ If the right wheel/servo does behave properly, then keep going.

```
/*
 * Robotics with the BOE Shield - RightServoTest
 * Right servo turns clockwise three seconds, stops 1 second, then
 * counterclockwise three seconds.
 */

#include <Servo.h>                      // Include servo library

Servo servoRight;                       // Declare right servo

void setup()                            // Built in initialization block
{
```

```
  servoRight.attach(12);                // Attach right signal to pin 12

  servoRight.writeMicroseconds(1300);   // Right wheel clockwise
  delay(3000);                          // ...for 3 seconds

  servoRight.writeMicroseconds(1500);   // Stay still
  delay(1000);                          // ...for 3 seconds

  servoRight.writeMicroseconds(1700);   // Right wheel counterclockwise
  delay(3000);                          // ...for 3 seconds

  servoRight.writeMicroseconds(1500);   // Right wheel counterclockwise

}

void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
}
```

Your Turn – Testing the Left Wheel

Now, it's time to run the same test on the left wheel as shown below. This involves modifying the RightServoTest sketch.



Clockwise 3 seconds *Stop 1 second* *Counterclockwise 3 seconds*

- ✓ Save RightServoTest as LeftServoTest.
- ✓ Change **`Servo servoRight`** to **`Servo servoLeft`**.
- ✓ Change **`servoRight.Attach(12)`** to **`servoLeft.Attach(13)`**.
- ✓ Replace the rest of the **`servoRight`** references with **`servoLeft`**.
- ✓ Save the modified sketch, and run it on your Arduino.
- ✓ Verify that it makes the left servo turn clockwise for 3 seconds, stop for 1 second, then turn counterclockwise for 3 seconds.
- ✓ If the left wheel does not behave as predicted, see Servo Troubleshooting below.

- ✓ If the left wheel/servo does behave properly, then your BOE Shield-Bot is functioning properly. You are ready to move on to Activity 3: Start-Reset Indicator on page 89.

Servo Troubleshooting

The servo doesn't turn at all.

- ✓ Make sure the battery pack has fresh batteries, all oriented properly in the case.
- ✓ Make sure the BOE Shield's power switch is set to position-2. Then re-run the program by pressing and releasing the board's RESET button.
- ✓ Double-check your servo connections. You may have inserted the plug over the pins backward, or offset it so it is only catching on one or two pins.
- ✓ Double-check your sketch against RightServoTest for the right servo, and double-check the modifications for the left servo.

The left servo turns when the right one is supposed to.

- ✓ Disconnect power—both battery pack and programming cable.
- ✓ Unplug both servos.
- ✓ Connect the servo that was connected to pin 12 to pin 13.
- ✓ Connect the other servo (that was connected to pin 13) to pin 12.
- ✓ Reconnect power, re-run the RightServoTest sketch to test the right servo, and then modify it to test the left servo.

The wheel does not fully stop; it still turns slowly.

- ✓ Try adjusting in hardware: Go back and re-do Chapter 2, Activity 5: Centering the Servos on page 63. If the servos are not mounted to give easy access to the potentiometer ports, use a long-shaft screwdriver or consider re-orienting them for re-assembly.
- ✓ Try adjusting in software: If the wheel turns slowly counterclockwise, use a value that's a little smaller than 1500. If it's turning clockwise, use a value that's a little larger than 1500. This new value will be used in place of 1500 for all `writeMicroseconds` calls for that wheel as you do the experiments in this book.

The wheel never stops, it just keeps turning rapidly.

- ✓ Try adjusting in hardware: Go back and re-do Chapter 2, Activity 5: Centering the Servos on page 63. If the servos are not mounted to give easy access to the potentiometer ports, use a long-shaft screwdriver or consider re-orienting them for re-assembly.
- ✓ If the servo speed and direction never varies while trying to center it, it may be damaged. Contact support@parallax.com for help.

Activity 3: Start-Reset Indicator

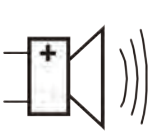
In this activity, we'll build a small noise-making circuit on the BOE Shield's prototyping area that will generate a tone if the robot's batteries run too low.

When the voltage supply drops below the level a device needs to function properly, it's called *brownout*. The device (your Arduino) typically shuts down until the supply voltage returns to normal. Then, it will restart whatever sketch it was running.

Brownouts typically happen when batteries are already running low, and the servos suddenly demand more power. For example, if the BOE Shield-Bot changes from full speed forward to full speed backward, the servos have to do extra work to stop the servos and then go the other direction. For this, they need more current, and when they try to pull that current from tired batteries, the output voltage dips enough to cause brownout.

Now, imagine your BOE Shield-Bot is navigating through a routine, and suddenly it stops for a moment and then goes in a completely unexpected direction. How will you know if it is a mistake in your code, or if it's a brownout? One simple, effective solution is to add a speaker to your BOE Shield-Bot and make it play a "start" tone at the beginning of every sketch. That way, if your BOE Shield-Bot has a brownout while it's navigating, you'll know right away because it'll play the start tone.

We'll use a device called a *piezoelectric* speaker (piezospeaker) that can make different tones depending on the frequency of high/low signals it receives from the Arduino. The schematic symbol and part drawing are shown below.



Frequency is the measurement of how often something occurs in a given amount of time.

A piezoelectric element is a crystal that changes shape slightly when voltage is applied to it. Applying high and low voltages at a rapid rate causes the crystal to rapidly change shape. The resulting vibration in turn vibrates the air around it, and this is what our ear detects as a tone. Every rate of vibration makes a different tone.

Piezoelectric elements have many uses. When force is applied to a piezoelectric element, it can create voltage. Some piezoelectric elements have a frequency at which they naturally vibrate. These can be used to create voltages at frequencies that function as the clock oscillator for many computers and microcontrollers.

Build the Piezospeaker Circuit

Parts Required

(1) piezospeaker (just peel off the “Remove the seal after washing” sticker if it has one)
(misc.) jumper wires

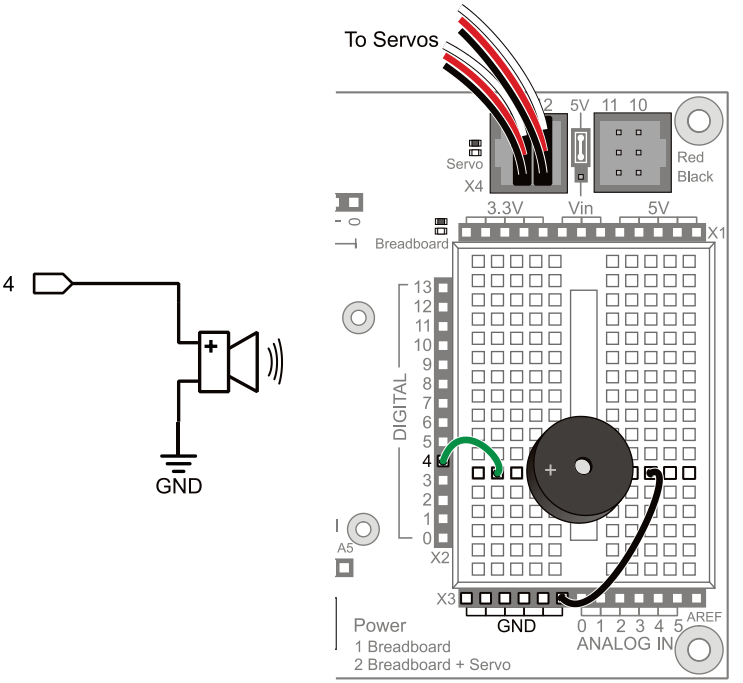
Building the Start/Reset Indicator Circuit

Always disconnect power before building or modifying circuits!

- Set the Power switch to 0.
- Unplug the battery pack.
- Unplug the programming cable.

The next picture shows a wiring diagram for adding a piezospeaker to the breadboard.

✓ Build the circuit shown next. Position the piezospeaker just as shown on the breadboard. The speaker should stay in place for the rest of the book, while other circuits are added or removed around it.



Programming the Start-Reset Indicator

The next example sketch tests the piezospeaker using calls to the Arduino's **tone** function. True to its name, this function sends signals to speakers to make them play tones.

There are two options for calling the **tone** function. One allows you to specify the ***pin*** and ***frequency*** (pitch) of the tone. The other allows you to specify ***pin***, ***frequency***, and ***duration*** (in milliseconds). We'll be using the second option since we don't need the tone to go on indefinitely.

```
tone(pin, frequency)
tone(pin, frequency, duration)
```

This piezospeaker is designed to play 4.5 kHz tones for smoke alarms, but it can also play a variety of audible tones and usually sounds best in the 1 kHz to 3.5 kHz range. The start-alert tone we'll use is:

```
tone(4, 3000, 1000);
delay(1000);
```

That will make pin 4 send a series of high/low signals repeating at 3 kHz (3000 times per second). The tone will last for 1000 ms, which is 1 second. The **tone** function continues in the background while the sketch moves on to the next command. We don't want the servos to start moving until the tone is done playing, so the **tone** command is followed by **delay(1000)** to let the tone finish before the sketch can move on to servo control.

Frequency can be measured in hertz (Hz) which is the number of times a signal repeats itself in one second. The human ear is able to detect frequencies in a range from very low pitch (20 Hz) to very high pitch (20 kHz or 20,000 Hz). One kilohertz (kHz) is one-thousand-times-per-second.

Example Sketch: StartResetIndicator

This example sketch makes a beep when it starts running, then it goes on to send the Serial Monitor messages every half-second. These messages will continue indefinitely because they are in the **loop** function. If the power to the Arduino is interrupted, the sketch will start at the beginning again, and you will hear the beep.

- ✓ Reconnect power to your board.
- ✓ Create and save the StartResetIndicator sketch, and run it on the Arduino.
- ✓ If you did not hear a tone, check your wiring and code for errors and try again.
- ✓ If you did hear an audible tone, open the Serial Monitor (this may cause a reset too). Then, push the reset button on the BOE Shield.

- ✓ Verify that, after each reset, the piezospeaker makes a clearly audible tone for one second, and then the "Waiting for reset…" messages resumes.
- ✓ Try disconnecting and reconnecting your battery supply and programming cable, and then plugging them back in. This should also trigger the start-alert tone.

```
/*
 * Robotics with the BOE Shield - StartResetIndicator
 * Test the piezospeaker circuit.
 */

void setup()                                // Built in initialization block
{
  Serial.begin(9600);
  Serial.println("Beep!");

  tone(4, 3000, 1000);                      // Play tone for 1 second
  delay(1000);                              // Delay to finish tone
}

void loop()                                 // Main loop auto-repeats
{
  Serial.println("Waiting for reset...");
  delay(1000);
}
```

How StartResetIndicator Works

StartResetIndicator begins by displaying the message "Beep!" in the Serial Monitor. Then, immediately after printing the message, the **tone** function plays a 3 kHz tone on the piezoelectric speaker for 1 second. Because the instructions are executed so rapidly by the Arduino, it should seem as though the message is displayed at the same instant the piezospeaker starts to play the tone.

When the tone is done, the sketch enters the **loop** function, which displays the same "Waiting for reset…" message over and over again. Each time the reset button on the BOE Shield is pressed or the power is disconnected and reconnected, the sketch starts over again with the "Beep!" message and the 3 kHz tone.

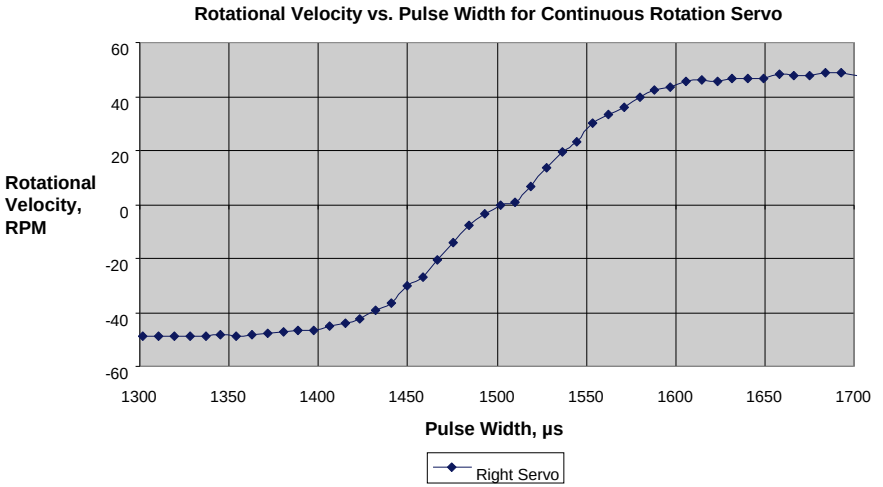
Your Turn – Adding StartResetIndicator to a Different Sketch

We'll use **tone** at the beginning of every example sketch from here onward. So, it's a good idea to get in the habit of putting **tone** and **delay** statements at the beginning of every sketch's **setup** function.

- ✓ Copy the **tone** and **delay** function calls from the StartResetIndicator sketch into the beginning of the RightServoTest sketch's **setup** function.
- ✓ Run the modified sketch and verify that it responds with the "sketch starting" tone every time the Arduino is reset.

Activity 4: Test Speed Control

The transfer curve graph on the next page shows pulse time vs. servo speed. The graph's horizontal axis shows the pulse width in microseconds (µs), and the vertical axis shows the servo's response in RPM. Clockwise rotation is shown as negative, and counterclockwise is positive. This particular servo's graph, which can also be called a *transfer curve*, ranges from about -48 to +48 RPM over the range of test pulse widths from 1300 to 1700 µs. A transfer curve graph of your servos would be similar.



Three Reasons Why the Transfer Curve Graph is Useful

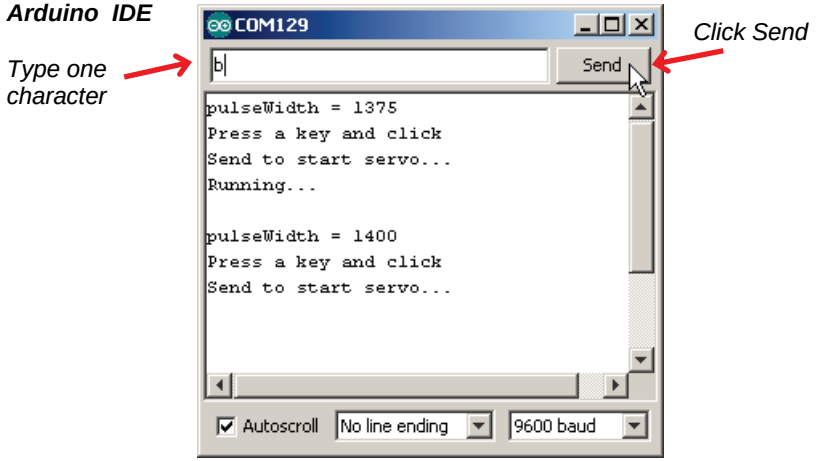
1. You can get a good idea of what to expect from your servo for a certain pulse width. Follow the vertical line up from 1500 to where the graph crosses it, then follow the horizontal line over and you'll see that there is zero rotation for 1500 µs pulses. We already knew **`servoLeft.writeMicroseconds(1500)`** stops a servo, but try some other values.
    - Compare servo speeds for 1300 and 1350 µs pulses. Does it really make any difference?
    - What speed would you expect from your servos with 1550 µs pulses? How about 1525 µs pulses?
2. Speed doesn't change much between 1300 and 1400 µs pulses. So, 1300 µs pulses for full speed clockwise is overkill; the same applies to 1600 vs. 1700 µs pulses for counterclockwise rotation. These overkill speed settings are useful because they are

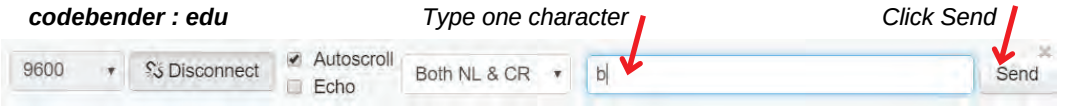
more likely to result in closely matched speeds than picking two values in the 1400 to 1600 µs range.

3. Between 1400 and 1600 µs, speed control is more or less linear. In this range, a certain change in pulse width will result in a corresponding change in speed. Use pulses in this range to control your servo speeds.

Example Sketch: TestServoSpeed

With this sketch, you can check servo RPM speed (and direction) for pulse values from 1375 µs to 1625 µs in steps of 25 µs. These speed measurements will help make it clear how servo control pulse durations in the 1400 to 1600 µs range control servo speed. This sketch starts by displaying the pulse duration that it's ready to send as a servo control signal. Then, it waits for you to send the Arduino a character with the Serial Monitor before it runs the servo. It runs the servo for six seconds, and during that time you can count the number of full turns the wheel makes. After that, the **`for`** loop repeats itself, and increases the pulse duration by 25 for the next test.





- ✓ Place a mark (like a piece of masking tape) on the wheel so that you can see how many revolutions it turns during the wheel speed tests.
- ✓ Set the BOE Shield-Bot on its nose so that the wheels can spin freely.
- ✓ Create and save the TestServoSpeed sketch, and run it on the Arduino.
- ✓ Open the Serial Monitor. Set the menus to 9600 baud and "no line ending."

- ✓ Click the transmit pane at the top, type any character, and click the Send button.
- ✓ Count the number of turns the wheel makes, and multiply by 10 for RPMs. (Don't forget to make a note of direction; it will change after the 5th test.)

If you were to add your data points to the graph, would they fit the overall shape?

```
/*
 Robotics with the BOE Shield – TestServoSpeed
 Send a character from the Serial Monitor to the Arduino to make the
 left servo run for 6 seconds.  Starts with 1375 us pulses and increases by
 25 us with each repetition, up to 1625 us.  This sketch is useful for
 graphing speed vs. pulse width.
 */

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left servo signal
Servo servoRight;                       // Declare right servo signal

void setup()                            // Built in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  Serial.begin(9600);                   // Set data rate to 9600 bps
  servoLeft.attach(13);                 // Attach left signal to P13
}

void loop()                             // Main loop auto-repeats
{
  // Loop counts with pulseWidth from 1375 to 1625 in increments of 25.

  for(int pulseWidth = 1375; pulseWidth <= 1625; pulseWidth += 25)
  {
    Serial.print("pulseWidth = ");           // Display pulseWidth value
    Serial.println(pulseWidth);
    Serial.println("Press a key and click"); // User prompt
    Serial.println("Send to start servo...");

    while(Serial.available() == 0);         // Wait for character
    Serial.read();                          // Clear character

    Serial.println("Running...");
    servoLeft.writeMicroseconds(pulseWidth); // Pin13 servo speed = pulse
    delay(6000);                            // ..for 6 seconds
    servoLeft.writeMicroseconds(1500);      // Pin 13 servo speed = stop
  }
}
```

How TestServoSpeed Works

The sketch TestServoSpeed increments the value of a variable named **pulseWidth** by 25 each time through a **for** loop.

```
// Loop counts with pulseWidth from 1375 to 1625 in increments of 25.

for(int pulseWidth = 1375; pulseWidth <= 1625; pulseWidth += 25)
```

With each repetition of the **for** loop, it displays the value of the next pulse width that it will send to the pin 13 servo, along with a user prompt.

```
  Serial.print("pulseWidth = ");           // Display pulseWidth value
  Serial.println(pulseWidth);
  Serial.println("Press a key and click"); // User prompt
  Serial.println("Send to start servo...");
```

After **Serial.begin** in the **setup** loop, the Arduino sets aside some memory for characters coming in from the Serial Monitor. This memory is typically called a *serial buffer*, and that's where ASCII values from the Serial Monitor are stored. Each time you use **Serial.read** to get a character from the buffer, the Arduino subtracts 1 from the number of characters waiting in the buffer.

A call to **Serial.available** will tell you how many characters are in the buffer. This sketch uses **while(Serial.available() = = 0)** to wait until the Serial Monitor sends a character. Before moving on to run the servos, it uses **Serial.read( )** to remove the character from the buffer. The sketch could have used **int myVar = Serial.read( )** to copy the character to a variable. Since the code isn't using the character's value to make decisions, it just calls **Serial.read**, but doesn't copy the result anywhere. The important part is that it needs to clear the buffer so that **Serial.available( )** returns zero next time through the loop.

```
while(Serial.available() == 0);      // Wait for character
Serial.read();                       // Clear character
```

Where is the `while` loop's code block? The C language allows the **while** loop to use an empty code block, in this case to wait there until it receives a character. When you type a character into the Serial Monitor, **Serial.available** returns 1 instead of 0, so the **while** loop lets the sketch move on to the next statement. **Serial.read** removes that character you typed from the Arduino's serial buffer to make sure that **Serial.available** returns 0 next time through the loop. You could have typed this empty while loop other ways:

```
while(Serial.available() == 0) {}
```

...or:

```
while(Serial.available() == 0) {};
```

After the Arduino receives a character from the keyboard, it displays the "Running..." message and then makes the servo turn for 6 seconds. Remember that the **for** loop this code is in starts the **pulseWidth** variable at 1375 and adds 25 to it with each repetition. So, the first time through the loop, **servoLeft** is 1375, the second time through it's 1400, and so on all the way up to 1625.

Each time through the loop, **servoLeft.writeMicroseconds(pulseWidth)** uses the value that **pulseWidth** stores to set servo speed. That's how it updates the servo's speed each time you send a character to the Arduino with the Serial Monitor.

```
Serial.println("Running...");
servoLeft.writeMicroseconds(pulseWidth); // Pin 13 speed=pulse
delay(6000);                             // ..for 6 seconds
servoLeft.writeMicroseconds(1500);       // Pin 13 speed=stop
```

Optional: Record Your Own Transfer Curve Data

You can use the table below to record the data for your own transfer curve. The loop in the TestServoSpeed sketch can be modified to test every value in the table, or every other value to save time.

- ✓ Change the **for** statement in TestServoSpeed from:

```
for(int pulseWidth=1375; pulseWidth <= 1625; pulseWidth += 25)
```

...to:

```
for(int pulseWidth=1300; pulseWidth <= 1700; pulseWidth += 20)
```

- ✓ Load the modified sketch into the Arduino and use it to fill in every other table entry. (If you want to fill in every table entry, **use pulseWidth += 10** in the for statement's ***increment*** parameter.)
- ✓ Use graphing software of your choice to plot the pulse width vs. wheel RPM.
- ✓ To repeat these measurements for the right wheel, replace all instances of 13 with 12 in the sketch.

Pulse Width and RPM for Parallax Continuous Rotation Servo							
Pulse Width (µs)	**Rotational Velocity (RPM)**	**Pulse Width (µs)**	**Rotational Velocity (RPM)**	**Pulse Width (µs)**	**Rotational Velocity (RPM)**	**Pulse Width (µs)**	**Rotational Velocity (RPM)**
1300		1400		1500		1600	
1310		1410		1510		1610	
1320		1420		1520		1620	
1330		1430		1530		1630	
1340		1440		1540		1640	
1350		1450		1550		1650	
1360		1460		1560		1660	
1370		1470		1570		1670	
1380		1480		1580		1680	
1390		1490		1590		1690	
						1700	

Chapter 3 Summary

This chapter covered BOE Shield-Bot assembly and testing. Assembly involved both mechanical construction and circuit-building, and testing used some new programming concepts. Here are the highlights:

Hardware Setup

- Mounting the servos and battery pack on the robot chassis
- Attaching the wheels to the servo motors, and the tail wheel to the chassis
- Mounting the BOE Shield with the Arduino onto the chassis

Electronics

- What a piezoelectric speaker is, and how to add one to the BOE Shield-Bot's breadboard circuits
- What frequency is, the units for measuring it, and what frequency range is audible to human hearing
- What a low-battery brownout condition is, and how certain servo maneuvers can cause it

Programming

- How to use the Arduino's **`tone`** function to play beeps on a piezospeaker
- What a serial buffer is, and how to use it with the Arduino's **`Serial.available`** function
- How to use the transmit pane of the Serial Monitor, and the **`Serial.read`** function, to send data to the Arduino
- How to use a **`for`** loop with an incrementing variable to change a servo's RPM
- How to add **`tone`** function calls to a sketch to audibly indicate the completion of a programming task

Engineering Skills

- Implementing a brownout indicator strategy to help determine if unexpected robot behavior is due to a low-battery condition, and not a programming error
- What a transfer curve graph is, and how it can be useful to understand the servo control signal vs. servo RPM relationship

Chapter 3 Challenges

Questions

1. What are some of the symptoms of brownout on the BOE Shield-Bot?
2. What is a reset?
3. How can a piezospeaker be used to announce that brownout just occurred?
4. What function makes the speaker play tones?
5. What's a hertz? What's its abbreviation?

Exercises

1. Write a statement that makes a tone, one that sounds different from the start-alert tone, to signify the end of a sketch.
2. Write a statement that plays a speaker tone to signify an intermediate step in the sketch. This tone should be different from a start-alert or end tone.

Projects

1. Modify the TestServoSpeed sketch so that it makes a tone signifying each test is complete.
2. Modify the TestServoSpeed sketch so that it runs both wheels instead of just one with each test. Make the right wheel turn the opposite direction from the left wheel.

Question Solutions

1. Symptoms include erratic behavior such as going in unexpected directions or doing a confused dance.
2. It's when the Arduino restarts executing a sketch from the beginning. Resets occur when you press/release the reset button, disconnect/reconnect power, or when the Arduino receives insufficient power due to brownout.
3. If you add statements that make the piezospeaker play a tone at the beginning of all your sketches, it'll play a tone if a brownout occurs. That way, you can know whether to replace the batteries or check for an error in your navigation code.
4. The **`tone`** function.
5. A hertz is a measurement of the number of times per second a signal repeats itself. It is abbreviated Hz.

Exercise Solutions

1. `tone(4, 2000, 1500);  //example, your tone may be different.`
2. `tone(4, 4000, 75);  //example, your tone may be different.`

Project Solutions

1. Add E2 solution to the end of the **`for`** loop.

```
/*
 Robotics with the BOE Shield – Chapter 3, Project 1
*/

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left servo signal
Servo servoRight;                       // Declare right servo signal

void setup()                            // Built in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  Serial.begin(9600);                   // Set data rate to 9600 bps
  servoLeft.attach(13);                 // Attach left signal to P13
}

void loop()                             // Main loop auto-repeats
{

  // Loop counts with pulseWidth from 1375 to 1625 in increments of 25.

  for(int pulseWidth = 1375; pulseWidth <= 1625; pulseWidth += 25)
  {
    Serial.print("pulseWidth = ");           // Display pulseWidth value
    Serial.println(pulseWidth);
    Serial.println("Press a key and click"); // User prompt
    Serial.println("Send to start servo...");

    while(Serial.available() == 0);         // Wait for character
    Serial.read();                          // Clear character

    Serial.println("Running...");
    servoLeft.writeMicroseconds(pulseWidth); // Pin13 servo speed = pulse
    delay(6000);                            // ..for 6 seconds
    servoLeft.writeMicroseconds(1500);      // Pin13 servo speed = stop
    tone(4, 4000, 75);                      // Test complete
  }
}
```

2. Add **`Servo servoRight`**, and **`servoRight.attach(12)`**. For same speed in opposite direction use:

```
servoRight.writeMicroseconds(1500 + (1500 – pulseWidth))
```

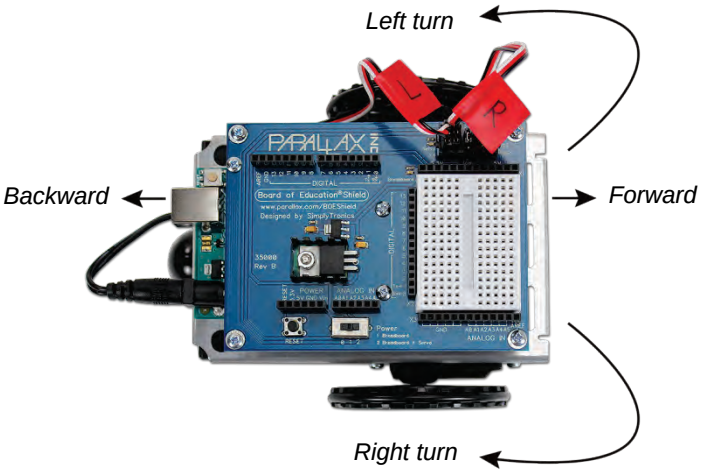
Remember to add a **`servoRight.writeMicroseconds(1500)`** after the 6-second run time.

```
/*
 Robotics with the BOE Shield – Chapter 3, Project 2
*/

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left servo signal
Servo servoRight;                       // Declare right servo signal

void setup()                            // Built in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  Serial.begin(9600);                   // Set data rate to 9600 bps
  servoLeft.attach(13);                 // Attach left signal to P13
  servoRight.attach(12);                // Attach right signal to P12
}

void loop()                             // Main loop auto-repeats
{

  // Loop counts with pulseWidth from 1375 to 1625 in increments of 25.

  for(int pulseWidth = 1375; pulseWidth <= 1625; pulseWidth += 25)
  {
    Serial.print("pulseWidth = ");           // Display pulseWidth value
    Serial.println(pulseWidth);
    Serial.println("Press a key and click"); // User prompt
    Serial.println("Send to start servo...");

    while(Serial.available() == 0);         // Wait for character
    Serial.read();                          // Clear character

    Serial.println("Running...");
    servoLeft.writeMicroseconds(pulseWidth); // Pin13 servo speed = pulse
    // Pin 12 servo opposite direction of pin 13 servo.
    servoRight.writeMicroseconds(1500 + (1500 - pulseWidth));
    delay(6000);                            // ..for 6 seconds
    servoLeft.writeMicroseconds(1500);      // Pin 13 servo speed = stop
    servoRight.writeMicroseconds(1500);     // Pin 12 servo speed = stop
    tone(4, 4000, 75);                      // Test complete
  }
}
```

Chapter 4. BOE Shield-Bot Navigation

This chapter introduces different programming strategies to make the BOE Shield-Bot move. Once we understand how basic navigation works, we'll make functions for each maneuver. In later chapters, we'll write sketches that call these functions in response to sensor input, which will allow the BOE Shield-Bot to navigate on its own.

The first step is to get oriented! The picture below shows forward, backward, left turn, and right turn from the point of view of the BOE Shield-Bot.

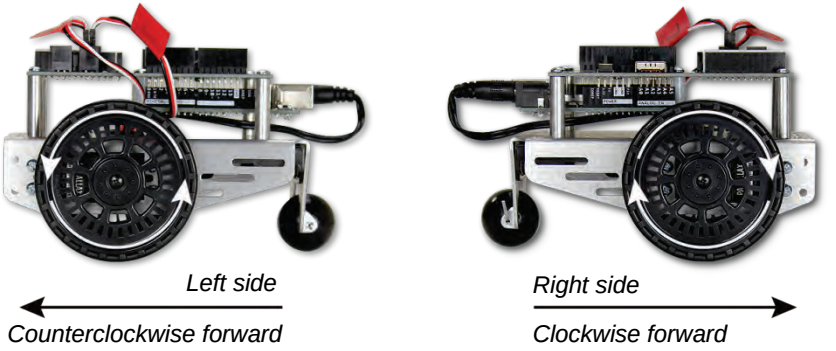


Here are the goals of this chapter's activities:

- Perform basic maneuvers: forward, backward, rotate left, rotate right, pivoting turns to the left and right, and stop.
- Tune the maneuvers from Activity #1 so that they are more precise.
- Use math to calculate servo run times to make the BOE Shield-Bot travel a predetermined distance.
- Write sketches that make the BOE Shield-Bot gradually accelerate into and decelerate out of maneuvers.
- Write functions to perform the basic maneuvers, and incorporate function calls into the sketches wherever they're needed.
- Design a function for quickly defining custom maneuvers.
- Store complex maneuvers in arrays and write sketches that play back these maneuvers.

Activity 1: Basic BOE Shield-Bot Maneuvers

Have you ever thought about what direction a car's wheels have to turn to propel it forward? The wheels turn opposite directions on opposite sides of the car. Likewise, to make the BOE Shield-Bot go forward, its left wheel has to turn counterclockwise, but its right wheel has to turn clockwise.



Remember that a sketch can use the Servo library's **`writeMicroseconds`** function to control the speed and direction of each servo. Then, it can use the **`delay`** function to keep the servos running for certain amounts of time before choosing new speeds and directions. Here's an example that will make the BOE Shield-Bot roll forward for about three seconds, and then stop.

Example Sketch: ForwardThreeSeconds

- ✓ Make sure the BOE Shield's power switch is set to 1 and the battery pack is plugged into the Arduino.
- ✓ Create and save the ForwardThreeSeconds sketch, and run it on the Arduino.
- ✓ Disconnect the programing cable and put the BOE Shield-Bot on the floor.

While holding down the Reset button, move the switch to position 3, and then let go. The BOE Shield-Bot should drive forward for three seconds.

```
// Robotics with the BOE Shield - ForwardThreeSeconds
// Make the BOE Shield-Bot roll forward for three seconds, then stop.

#include <Servo.h>                    // Include servo library

Servo servoLeft;                      // Declare left and right servos
Servo servoRight;
```

```
void setup()                              // Built-in initialization block
{
  tone(4, 3000, 1000);                    // Play tone for 1 second
  delay(1000);                            // Delay to finish tone

  servoLeft.attach(13);                   // Attach left signal to pin 13
  servoRight.attach(12);                  // Attach right signal to pin 12

  // Full speed forward
  servoLeft.writeMicroseconds(1700);      // Left wheel counterclockwise
  servoRight.writeMicroseconds(1300);     // Right wheel clockwise
  delay(3000);                            // ...for 3 seconds

  servoLeft.detach();                     // Stop sending servo signals
  servoRight.detach();
}

void loop()                               // Main loop auto-repeats
{                                         // Empty, nothing needs repeating
}
```

How ForwardThreeSeconds Works

First, the Servo library has to be included so that your sketch can access its functions:

```
#include <Servo.h>              // Include servo library
```

Next, an instance of **`Servo`** must be declared and uniquely named for each wheel:

```
Servo servoLeft;                // Declare left & right servos
Servo servoRight;
```

Instance of an Object: An *object* is a block of pre-written code that can be copied and re-used multiple times in a single sketch. Each copy, called an object instance, can be configured differently. For example, the two **`Servo`** declarations create two instances of the object's code, named **`servoLeft`** and **`servoRight`**. Then, functions within each instance can be called and configured individually. So, **`servoLeft.attach(13)`** configures the **`servoLeft`** object instance to send its servo control signals to pin 13. Likewise, **`servoRight.attach(12)`** tells the **`servoRight`** object instance to send its signals to pin 12.

A sketch automatically starts in its **`setup`** function. It runs the code in there once before moving on to the **`loop`** function, which automatically keeps repeating. Since we only want the BOE Shield-Bot to go forward and stop once, all the code can be placed in the **`setup`** function. This leaves the **`loop`** function empty, but that's okay.

As with all motion sketches, the first action **setup** takes is making the piezospeaker beep. The **tone** function call transmits a signal to digital pin 4 that makes the piezospeaker play a 3 kHz tone that lasts for 1 second. Since the **tone** function works in the background while the code moves on, **delay(1000)** prevents the BOE Shield-Bot from moving until the tone is done playing.

```
void setup()                              // Built-in initialization
  {
    tone(4, 3000, 1000);                  // Play tone for 1 second
    delay(1000);                          // Delay to finish tone
```

Next, the **servoLeft** object instance gets attached to digital pin 13 and the **servoRight** instance gets attached to pin 12. This makes calls to **servoLeft.writeMicroseconds** affect the servo control signals sent on pin 13. Likewise, **servoRight.writeMicroseconds** calls will affect the signals sent on pin 12.

```
    servoLeft.attach(13);                 // Attach left signal to pin 13
    servoRight.attach(12);                // Attach right signal to pin 12
```

Remember that we need the BOE Shield-Bot's left and right wheels to turn in opposite directions to drive forward. The function call **servoLeft.writeMicroseconds(1700)** makes the left servo turn full speed counterclockwise, and the function call **servoRight.writeMicroseconds(1300)** makes the right wheel turn full speed clockwise. The result is forward motion. The **delay(3000)** function call keeps the servos running at that speed for three full seconds. After the delay, s**ervoLeft.detach** and s**ervoRight.detach** discontinue the servo signals, which bring the robot to a stop.

```
   // Full speed forward
   servoLeft.writeMicroseconds(1700);  // Left wheel counterclockwise
   servoRight.writeMicroseconds(1300); // Right wheel clockwise
   delay(3000);                        // ...for 3 seconds

   servoLeft.detach();                 // Stop sending servo signals
   servoRight.detach();
}
```

After the **setup** function runs out of code, the sketch automatically advances to the **loop** function, which repeats itself indefinitely. In this case, we are leaving it empty because the sketch is done, so it repeats nothing, over and over again, indefinitely.

```
void loop()                   // Main loop auto-repeats
{
}
```

Your Turn – Adjusting Distance

Want to change the distance traveled? Just change the time in **`delay(3000)`**. For example, **`delay(1500)`** will make the BOE Shield-Bot go for only half the time, which in turn will make it travel only half as far. Likewise, **`delay(6000)`** will make it go for twice the time, and therefore twice the distance.

- ✓ Change **`delay(3000)`** to **`delay(1500)`** and re-load the sketch. Did the BOE Shield-Bot travel only half the distance?

Moving Backward, Rotating, and Pivoting

All it takes to get other motions out of your BOE Shield-Bot are different combinations of ***`us`*** parameters in your **`servoLeft`** and **`servoRight writeMicroseconds`** calls. For example, these two calls will make your BOE Shield-Bot go backwards:

```
// Full speed backwards
servoLeft.writeMicroseconds(1300);   // Left wheel clockwise
servoRight.writeMicroseconds(1700);  // Right wheel counterclockwise
```

These two calls will make your BOE Shield-Bot rotate in place to make a left turn:

```
// Turn left in place
servoLeft.writeMicroseconds(1300);   // Left wheel clockwise
servoRight.writeMicroseconds(1300);  // Right wheel clockwise
```

These two calls will make your BOE Shield-Bot rotate in place for a right turn:

```
// Turn right in place
servoLeft.writeMicroseconds(1700);   // Left wheel counterclockwise
servoRight.writeMicroseconds(1700); // Right wheel counterclockwise
```

Let's combine all these commands into a single sketch that makes the BOE Shield-Bot move forward, turn left, turn right, then move backward.

Example Sketch: ForwardLeftRightBackward

- ✓ Create, save, and run the ForwardLeftRightBackward sketch.
- ✓ Verify that the BOE Shield-Bot makes the forward-left-right-backward motions.

```
// Robotics with the BOE Shield - ForwardLeftRightBackward
// Move forward, left, right, then backward for testing and tuning.

#include <Servo.h>                     // Include servo library

Servo servoLeft;                       // Declare left and right servos
```

```
Servo servoRight;

void setup()                              // Built-in initialization block
{
  tone(4, 3000, 1000);                    // Play tone for 1 second
  delay(1000);                            // Delay to finish tone

  servoLeft.attach(13);                   // Attach left signal to pin 13
  servoRight.attach(12);                  // Attach right signal to pin 12

  // Full speed forward
  servoLeft.writeMicroseconds(1700);      // Left wheel counterclockwise
  servoRight.writeMicroseconds(1300);     // Right wheel clockwise
  delay(2000);                            // ...for 2 seconds

  // Turn left in place
  servoLeft.writeMicroseconds(1300);      // Left wheel clockwise
  servoRight.writeMicroseconds(1300);     // Right wheel clockwise
  delay(600);                             // ...for 0.6 seconds

  // Turn right in place
  servoLeft.writeMicroseconds(1700);      // Left wheel counterclockwise
  servoRight.writeMicroseconds(1700);     // Right wheel counterclockwise
  delay(600);                             // ...for 0.6 seconds

  // Full speed backward
  servoLeft.writeMicroseconds(1300);      // Left wheel clockwise
  servoRight.writeMicroseconds(1700);     // Right wheel counterclockwise
  delay(2000);                            // ...for 2 seconds

  servoLeft.detach();                     // Stop sending servo signals
  servoRight.detach();
}

void loop()                               // Main loop auto-repeats
{                                         // Empty, nothing needs repeating
}
```

QUICK TIP — To enter this sketch quickly, copy and paste to make four copies of the four lines that make up a maneuver. Then, modify each one with individual values.

Your Turn – Pivoting

You can make the BOE Shield-Bot turn by pivoting around one wheel. The trick is to keep one wheel still while the other rotates. Here are the four routines for forward and backward pivot turns:

```
 // Pivot forward-left
 servoLeft.writeMicroseconds(1500);   // Left wheel stop
 servoRight.writeMicroseconds(1300);  // Right wheel clockwise
```

```
// Pivot forward-right
servoLeft.writeMicroseconds(1700);   // Left wheel counterclockwise
servoRight.writeMicroseconds(1500);  // Right wheel stop

// Pivot backward-left
servoLeft.writeMicroseconds(1500);   // Left wheel stop
servoRight.writeMicroseconds(1700);  // Right wheel counterclockwise

// Pivot backward-right
servoLeft.writeMicroseconds(1300);   // Left wheel clockwise
servoRight.writeMicroseconds(1500);  // Right wheel stop
```

- ✓ Save ForwardLeftRightBackward as PivotTests.
- ✓ Change the ***us*** parameter in each **`writeMicroseconds`** call so the sketch will perform all four pivot maneuvers: forward, left, right, backward.
- ✓ Run the modified sketch and verify that the different pivot actions work.
- ✓ Try experimenting with the **`delay`** calls for each maneuver so that each one runs long enough to execute a 90° turn.

Activity 2: Tuning the Basic Maneuvers

Imagine writing a sketch that instructs your BOE Shield-Bot to travel full-speed forward for fifteen seconds. What if your robot curves slightly to the left or right during its travel, when it's supposed to be traveling straight ahead? There's no need to take the BOE Shield-Bot back apart and re-adjust the servos with a screwdriver to fix this. You can simply adjust the sketch slightly to get both wheels traveling the same speed. While the screwdriver approach could be considered a *hardware adjustment*, the programming approach would be a *software adjustment*.

Straightening the Shield-Bot's Path

So, would your BOE Shield-Bot travel in an arc instead of in a straight line? Top speed varies from one servo to the next, so one wheel is likely to rotate a little faster than the other, causing the BOE Shield-Bot to make a gradual turn.

To correct this, the first step is to examine your BOE Shield-Bot's forward travel for a longer period of time to see if it is curving, which way, and how much.

Example Sketch: ForwardTenSeconds

- ✓ Open ForwardThreeSeconds and re-save it as ForwardTenSeconds.
- ✓ Change **`delay(3000)`** to **`delay(10000)`**, and update the title and comments too.
- ✓ Set the power switch to 1, run the sketch, then disconnect the USB cable.

- ✓ Place the Shield-Bot on a long stretch of smooth, bare floor.
- ✓ Hold the Reset button down while you move the Power switch to 2, then let go.
- ✓ Press the Reset button, then watch closely to see if your BOE Shield-Bot veers to the right or left as it travels forward for ten seconds.

```
// Robotics with the BOE Shield - ForwardTenSeconds
// Make the BOE Shield-Bot roll forward for ten seconds, then stop.

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12

  // Full speed forward
  servoLeft.writeMicroseconds(1700);    // Left wheel counterclockwise
  servoRight.writeMicroseconds(1300);   // Right wheel clockwise
  delay(10000);                         // ...for 10 seconds

  servoLeft.detach();                   // Stop sending servo signals
  servoRight.detach();
}

void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
}
```

Your Turn – Adjusting Servo Speed to Straighten the BOE Shield-Bot's Path

If your BOE Shield-Bot turns slightly when you want it to go straight forward, the solution is fairly simple. Just slow down the faster wheel. Remember from the servo transfer curve graph that you have best speed control over the servos in the 1400 to 1600 µs range.

The *us* parameters in writeMicroseconds(*us*)				
Top speed clockwise	Linear speed zone starts	Full stop	Linear speed zone ends	Top speed counterclockwise
1300	1400	1500	1600	1700

Let's say that your BOE Shield-Bot gradually turns left. That means the right wheel is turning faster than the left. Since the left wheel is already going as fast as it possibly can, the right wheel needs to be slowed down to straighten out the robot's path. To slow it down, change the ***us*** parameter in **`servoRight.writeMicroseconds(us)`** to a value closer to 1500. First, try 1400. Is it still going too fast? Raise it 1410. Keep raising the parameter by 10 until the BOE Shield-Bot no longer curves to the left. If any adjustment overshoots 'straight' and your BOE Shield-Bot starts curving to the right instead, start decreasing the ***us*** parameter by smaller amounts. Keep refining that ***us*** parameter until your BOE Shield-Bot goes straight forward. This is called an *iterative process*, meaning that it takes repeated tries and refinements to get to the right value.

If your BOE Shield-Bot curved to the right instead of the left, it means you need to slow down the left wheel. You're starting with **`servoLeft.writeMicroseconds(1700)`** so the ***us*** parameter needs to decrease. Try 1600, then reduce by increments of 10 until it either goes straight or starts turning the other direction, and increase by 2 if you overshoot.

- ✓ Modify ForwardTenSeconds so that it makes your BOE Shield-Bot go straight forward.
- ✓ Use masking tape or a sticker to label each servo with the best ***us*** parameters for your **`writeMicroseconds(us)`** function calls.
- ✓ If your BOE Shield-Bot already travels straight forward, try the modifications just discussed anyway, to see the effect. It should cause the BOE Shield-Bot to travel in a curve instead of a straight line.

You might find that there's an entirely different situation when you program your BOE Shield-Bot to roll backward.

- ✓ Modify ForwardTenSeconds so that it makes the BOE Shield-Bot roll backward for ten seconds.
- ✓ Repeat the test for a straight line.
- ✓ Repeat the steps for correcting the ***us*** parameter for the **`writeMicroseconds`** function call to straighten the BOE Shield-Bot's backward travel.

Tuning the Turns

The amount of time the BOE Shield-Bot spends rotating in place determines how far it turns. So, to tune a turn, all you need to do is adjust the **`delay`** function's ***ms*** parameter to make it turn for a different amount of time.

Let's say that the BOE Shield-Bot turns just a bit more than 90° (1/4 of a full circle). Try **`delay(580)`**, or maybe even **`delay(560)`**. If it doesn't turn far enough, make it run longer by increasing the **`delay`** function's ***ms*** parameter 20 ms at a time.

The smallest change that actually makes a difference is 20. Servo control pulses are sent every 20 ms, so adjust your **`delay`** function call's ***`ms`*** parameter in multiples of 20.

If you find yourself with one value slightly overshooting 90° and the other slightly undershooting, choose the value that makes it turn a little too far, then slow down the servos slightly. In the case of rotating left, both **`writeMicroseconds`** ***`us`*** parameters should be changed from 1300 to something closer to 1500. Start with 1400 and then gradually increase the values to slow both servos. For rotating right, start by changing the ***`us`*** parameters from 1700 to 1600, and then experiment with reducing in increments of 10 from there.

Your Turn – 90° Rotating Turns and Sketch Updates

- ✓ Modify ForwardLeftRightBackward so that it makes precise 90° rotating turns.
- ✓ Update the label on each servo with a notation about the appropriate delay function ***`ms`*** parameter for a 90° turn.
- ✓ Update the **`delay`** function ***`ms`*** parameters in ForwardLeftRightBackward with the values that you determined for straight forward and backward travel.

Carpeting can cause navigation errors. If you are running your BOE Shield-Bot on carpeting, don't expect perfect results! The way the carpet pile is laying can affect the way your BOE Shield-Bot travels, especially over long distances. For more precise maneuvers, use a smooth surface.

Activity 3: Calculating Distances

In many robotics contests, more precise robot navigation means better scores. One popular entry-level robotics contest is called *dead reckoning*. The entire goal of this contest is to make your robot go to one or more locations and then return to exactly where it started.

You might remember asking your parents this question, over and over again, while on your way to a vacation destination or relatives' house: "Are we there yet?"

Perhaps when you got a little older, and learned division in school, you started watching the road signs to see how far it was to the destination city. Next, you checked the car's speedometer. By dividing the speed into the distance, you got a pretty good estimate of the time it would take to get there. You may not have been thinking in these exact terms, but here is the equation you were using:

$$time = \frac{distance}{speed}$$

U.S. customary units example: If you're 140 miles away from your destination, and you're traveling 70 miles per hour, it's going to take 2 hours to get there.

$$time = \frac{140\ miles}{70\ miles/hour}$$
$$= 140\ \ miles \times \frac{1\ hour}{70\ miles}$$
$$= 2\ hours$$

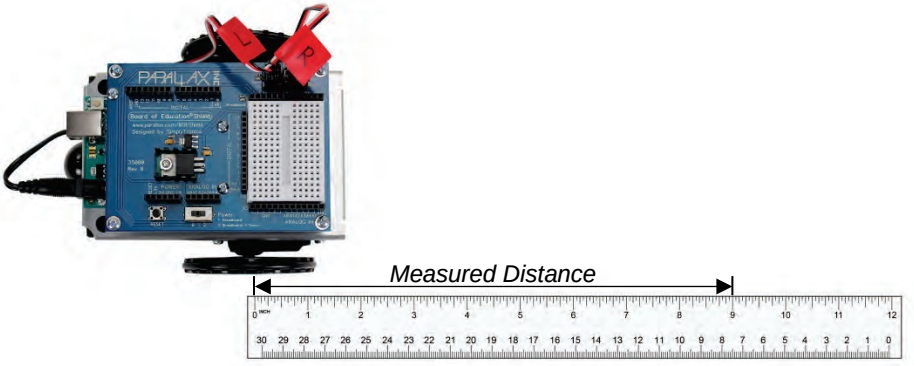
Metric units example: If you're 200 kilometers away from your destination, and you're traveling 100 kilometers per hour, it's going to take 2 hours to get there.

$$time = \frac{200\ kilometers}{100\ kilometers/hour}$$
$$= 200\ \ km \times \frac{1\ hour}{100\ km}$$
$$= 2\ hours$$

You can do the same exercise with the BOE Shield-Bot, except you have control over how far away the destination is. Here's the equation you will use:

$$servo\ run\ time = \frac{robot\ distance}{robot\ speed}$$

- ✓ Create, save, and run the sketch ForwardOneSecond.
- ✓ Place your BOE Shield-Bot next to a ruler.
- ✓ Make sure to line up the point where the wheel touches the ground with the 0 in/cm position on the ruler.



- ✓ Press the Reset button on your board to re-run the sketch.
- ✓ Measure how far your BOE Shield-Bot traveled by recording the measurement where the wheel is now touching the ground here:_____ (in or cm).

```
// Robotics with the BOE Shield - ForwardOneSecond
// Make the BOE Shield-Bot roll forward for one second, then stop.

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12
                                        // Full speed forward
  servoLeft.writeMicroseconds(1700);    // Left wheel counterclockwise
  servoRight.writeMicroseconds(1300);   // Right wheel clockwise
  delay(1000);                          // ...for 1 second

  servoLeft.detach();                   // Stop sending servo signals
  servoRight.detach();
}

void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
}
```

The distance you just recorded is your BOE Shield-Bot's speed, in units per second. Now, you can figure out how many seconds your BOE Shield-Bot has to travel to go a particular distance.

Inches and centimeters per second: The abbreviation for inches is in, and the abbreviation for centimeters is cm. Likewise, inches per second is abbreviated in/s, and centimeters per second is abbreviated cm/s. Both are convenient speed measurements for the BOE Shield-Bot. There are 2.54 cm in 1 in. You can convert inches to centimeters by multiplying the number of inches by 2.54. You can convert centimeters to inches by dividing the number of centimeters by 2.54

Keep in mind that your calculations will be in terms of seconds, but the **`delay`** function will need a parameter that's in terms of milliseconds. So, take your result, which is in terms of seconds, and multiply it by 1000. Then, use that value in your **`delay`** function call. For example, to make your BOE Shield-Bot run for 2.22 seconds, you'd use **`delay(2220)`** after your **`writeMicroseconds`** calls.

U.S. customary units example: At 9 in/s, your BOE Shield-Bot has to travel for 2.22 s to travel 20 in.

$$\begin{aligned} time &= \frac{20\ in}{9\ in/s} \\ &= 20\ in \times \frac{1\ s}{9\ in} \\ &= 2.22\ s \end{aligned}$$

Metric units example: At 23 cm/s, your BOE Shield-Bot has to travel for 2.22 s to travel 51 cm.

$$\begin{aligned} time &= \frac{51\ cm}{23\ cm/s} \\ &= 51\ cm \times \frac{1\ s}{23\ cm} \\ &= 2.22\ s \end{aligned}$$

Both examples above resolve to the same answer:

$$\begin{aligned} time(ms) &= time(s) \times \frac{1000\ ms}{s} \\ &= 2.22 s \times \frac{1000\ ms}{s} \\ &= 2220\ ms \end{aligned}$$

So, use **`delay(2220)`** after your **`writeMicroseconds`** calls:

```
servoLeft.writeMicroseconds(1700);
servoRight.writeMicroseconds(1300);
delay(2220);
```

Your Turn – Your BOE Shield-Bot's Distance

Now it's time to try this out with distances that you choose.

- ✓ Decide how far you want your BOE Shield-Bot to travel.
- ✓ Use the equation below to figure out how many milliseconds of forward travel you need for that distance:

$$ms\ servo\ run\ time = \frac{robot\ distance}{robot\ speed} \times \frac{1000\ ms}{s}$$

✓ Modify ForwardOneSecond to make your BOE Shield-Bot travel forward the amount of time that you determined, and try it out. How close does it come?

Encoders: Increase the accuracy of your BOE Shield-Bot distances with devices called encoders which count the holes in the BOE Shield-Bot's wheels as they pass.

Activity 4: Ramping Maneuvers

Ramping is a way to gradually increase or decrease the speed of the servos instead of abruptly starting or stopping. This technique can increase the life expectancy of both your BOE Shield-Bot's batteries and your servos.

Programming for Ramping

The diagram below shows an example of how to ramp up to full speed. The **for** loop declares an **int** variable named **speed**, and uses it to repeat the loop 100 times. With each repetition of the loop, the value of **speed** increases by 2 because of the **speed+=2** expression in the **for** loop's ***increment*** parameter. Since the **speed** variable is in each **writeMicroseconds** call's ***us*** parameter, it affects the value each time the **for** loop repeats. With the 20 ms delay between each repetition, the loop repeats at about 50 times per second. That means it takes **speed** 1 second to get to 100 in steps of 2, and at that point, both servos will be going about full speed.

```
for(int speed = 0; speed <= 100; speed+=2)
{
  servoLeft.writeMicroseconds(1500+speed);
  servoRight.writeMicroseconds(1500-speed);
  delay(20);
}
```

Let's take a closer look at the trips through the **for** loop from this diagram:

- First trip: **speed** is 0, so both **writeMicroseconds** calls end up with ***us*** parameters of 1500.
- Second trip: **speed** is 2, so we have **servoLeft.writeMicroseconds(1502)** and **servoRight.writeMicroseconds(1498)**.
- Third trip: **speed** is 4, so we have **servoLeft.writeMicroseconds(1504)** and **servoRight.writeMicroseconds(1496)**.

Keep on in this manner until the…

- 50th trip: **speed** is 100, with **servoLeft.writeMicroseconds(1600)** and **servoRight.writeMicroseconds(1400)**. Remember, that's pretty close to full speed; 1700 and 1300 are overkill.

Example Sketch: StartAndStopWithRamping

- ✓ Create, save, and run the sketch StartAndStopWithRamping.
- ✓ Verify that the BOE Shield-Bot gradually accelerates to full speed, maintains full speed for a while, and then gradually decelerates to a full stop.

```
// Robotics with the BOE Shield - StartAndStopWithRamping
// Ramp up, go forward, ramp down.

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12

  for(int speed = 0; speed <= 100; speed += 2) // Ramp up to full speed.
  {
    servoLeft.writeMicroseconds(1500+speed);   // us = 1500,1502,... 1600
    servoRight.writeMicroseconds(1500-speed);  // us = 1500,1498,... 1400
    delay(20);                                 // 20 ms at each speed
  }

  delay(1500);                           // Full speed for 1.5 seconds

  for(int speed = 100; speed >= 0; speed -= 2) // Ramp full speed to stop
  {
    servoLeft.writeMicroseconds(1500+speed);   // us = 1600,1598,... 1500
    servoRight.writeMicroseconds(1500-speed);  // us = 1400,1402,... 1500
    delay(20);                                 // 20 ms at each speed
  }

  servoLeft.detach();                    // Stop sending servo signals
  servoRight.detach();
}

void loop()                              // Main loop auto-repeats
{                                        // Empty, nothing to repeat
}
```

Your Turn – Add Ramping to Other Maneuvers

You can also create routines to combine ramping with other maneuvers. Here's an example of how to ramp up to full speed going backward instead of forward. The only difference between this routine and the forward ramping routine is that the value of **speed** starts at zero and counts to –100.

```
for(int speed = 0; speed >= -100; speed -= 2)// Ramp stop to reverse
{
  servoLeft.writeMicroseconds(1500+speed);   // us = 1500...1400
  servoRight.writeMicroseconds(1500-speed);  // us = 1500...1600
  delay(20);                                 // 20 ms at each speed
}
```

You can also make a routine for ramping into and out of a turn. Here is a right-turn ramping example. Notice that instead of **1500+speed** for one wheel and **1500–speed** for the other, now they are both **1500+speed**. For left-turn ramping, they would both be **1500–speed**.

```
for(int speed = 0; speed <= 100; speed += 2) // Ramp stop to right turn
{
  servoLeft.writeMicroseconds(1500+speed);   // us = 1500...1600
  servoRight.writeMicroseconds(1500+speed);  // us = 1500...1600
  delay(20);                                 // 20 ms at each speed
}

for(int speed = 100; speed >= 0; speed -= 2)// right turn to stop
{
  servoLeft.writeMicroseconds(1500+speed);   // us = 1600...1500
  servoRight.writeMicroseconds(1500+speed);  // us = 1600...1500
  delay(20);                                 // 20 ms at each speed
}
```

- ✓ Open the sketch ForwardLeftRightBackward and save it as ForwardLeftRightBackwardRamping.
- ✓ Modify the new sketch so your BOE Shield-Bot will ramp into and out of each maneuver. Hint: you might use the code snippets above, and similar snippets from StartAndStopWithRamping.

Activity 5: Simplify Navigation with Functions

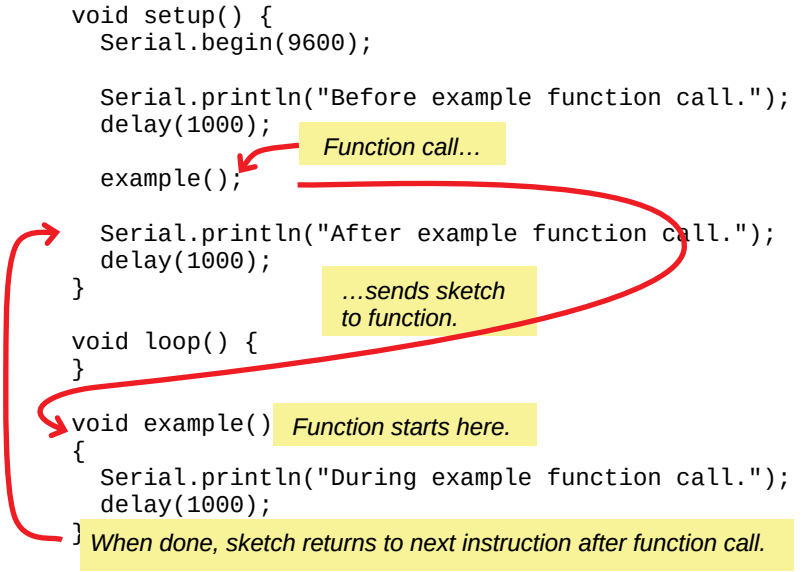
One convenient way to execute pre-programmed maneuvers is with functions. In the next chapter, your BOE Shield-Bot will have to perform maneuvers to avoid obstacles, and a key ingredient for avoiding obstacles is executing pre-programmed maneuvers.

The **setup** and **loop** functions are built into the Arduino language, but you can add more functions that do specific tasks for your sketch. This activity introduces how to add more

functions to your sketch as well as a few different approaches to creating reusable maneuvers with those functions.

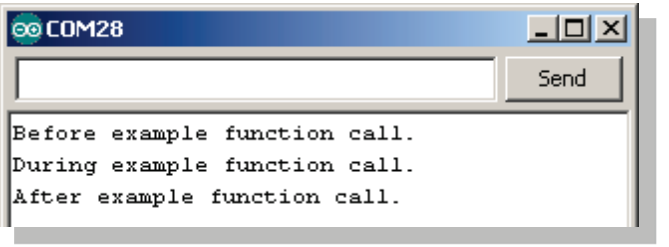
Minimal Function Call

The diagram below shows part of a sketch that contains a function named **example** added at the end, below the **loop** function. It begins and gets named with the function definition **void example()**. The empty parentheses means that it doesn't need any parameters to do its job, and **void** indicates that it does not return a value (we'll look at functions that return values in a later chapter). The curly braces **{}** that follow this definition contain the **example** function's block of code.



There is a function call to example in the **setup** function, labeled in the diagram above. That **example()** line tells the sketch to go find the function with that name, execute its code, and come back when done. So, the sketch jumps down to **void example()** and executes the two commands in its curly braces. Then, it returns to the function call and continues from there. Here is the order of events you will see when you run the sketch:

1. The Serial Monitor displays "Before example function call."
2. After a one second delay, the monitor displays "During example function call." Why? Because the **example()** call sends the code to **void example()**, which has the line of code that prints that message, followed by a 1 second delay. Then, the function returns to the **example** call in **setup**.
3. The Serial Monitor displays "After example function call."



Example Sketch – SimpleFunctionCall

- ✓ Create, save, and run the SimpleFunctionCall sketch.
- ✓ Watch the upload progress, and as soon as it's done, open the Serial Monitor.
- ✓ Watch your terminal and verify that the sequence of messages start with Before, then During, then After.

```
// Robotics with the BOE Shield – SimpleFunctionCall
// This sketch demonstrates a simple function call.

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

  Serial.println("Before example function call.");
  delay(1000);

  example();                            // This is the function call

  Serial.println("After example function call.");
  delay(1000);
}

void loop() {
}

void example()                          // This is the function
{
  Serial.println("During example function call.");
  delay(1000);
}
```

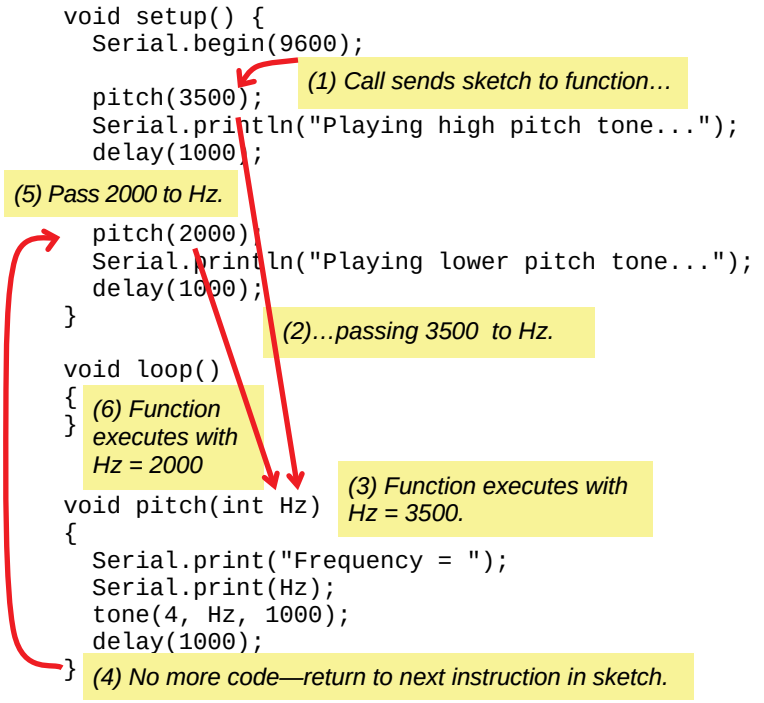
Function Call with Parameters

Remember that a function can have one or more *parameters*—data that the function receives and uses when it is called. The diagram below shows the **`pitch`** function from the next sketch. It is declared with **`void pitch(int Hz)`**. Recall that that the Arduino stores variable values in different data types, with **`int`** specifying an integer value in the range

of -32,768 to 32,767. Here, the term **`int Hz`** in the parentheses defines a parameter for the **`pitch`** function; in this case, it declares a local variable **`Hz`** of data type **`int`**.

> **Local variables**, remember, are declared within a function, and can only be seen and used inside that function. If a local variable is created as a parameter in the function declaration, as **`void pitch(int Hz)`** is here, initialize it by passing a value to it each time the function is called. For example, the call **`pitch(3500)`** passes the integer value 3500 to the **`pitch`** function's **`int Hz`** parameter.

So, when the first function call to **`pitch`** is made with **`pitch(3500)`**, the integer value 3500 gets passed to **`Hz`**. This initializes **`Hz`** to the value 3500, to be used during this trip through the **`pitch`** function's code block. The second call, **`pitch(2000)`**, initializes **`Hz`** to 2000 during the sketch's second trip through the **`pitch`** function's code block.



Example Sketch – FunctionCallWithParameter

- ✓ Read the sketch, and predict what you will see and hear when you run it.
- ✓ Create, save, and run FunctionCallWithParameter, then open the Serial Monitor.
- ✓ Watch your terminal and listen to the tones.

```
// Robotics with the BOE Shield – FunctionCallWithParameter
// This program demonstrates a function call with a parameter.

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

  Serial.println("Playing higher pitch tone...");

  pitch(3500);         // pitch function call passes 3500 to Hz parameter

  delay(1000);

  Serial.println("Playing lower pitch tone...");

  pitch(2000);         // pitch function call passes 2000 to Hz parameter

  delay(1000);
}

void loop()
{
}

void pitch(int Hz)     // pitch function with Hz declared as a parameter
{
  Serial.print("Frequency = ");
  Serial.println(Hz);
  tone(4, Hz, 1000);
  delay(1000);
}
```

Was your prediction correct? If so, great! If not, take a closer look at the sketch and make sure you can follow the code from each function call to the function and back. For clarification, take another look at the previous diagram.

Your Turn – Expand Function to Two Parameters

Here is a modified **`pitch`** function that accepts two parameters: **`Hz`** and **`ms`**. This new **`pitch`** function controls how long the tone lasts.

```
void pitch(int Hz, int ms)
{
  Serial.print("Frequency = ");
  Serial.println(Hz);
  tone(4, Hz, ms);
  delay(ms);
}
```

Here are two calls to the modified **`pitch`** function, one for a 0.5 second 3500 Hz tone, and the other for a 1.5 second 2000 Hz tone:

```
pitch(3500, 500);
pitch(2000, 1500);
```

Notice that each of these calls to **pitch** includes two values, one to pass to the **Hz** parameter, and one to pass to the **ms** parameter. The number of values in a function call must match the number of parameters in that function's definition, or the sketch won't compile.

- ✓ Save FunctionCallWithParameter as FunctionCallWithTwoParameters.
- ✓ Replace the **pitch** function with the two-parameter version.
- ✓ Replace the single parameter **pitch** calls with the two-parameter calls.
- ✓ Run the modified sketch and verify that it works.

Put Maneuvers Into Functions

Let's try putting the **forward**, **turnLeft**, **turnRight**, and **backward** navigation routines inside functions. Here's an example:

Example Sketch – MovementsWithSimpleFunctions

- ✓ Create, save, and run MovementsWithSimpleFunctions.

```
// Robotics with the BOE Shield - MovementsWithSimpleFunctions
// Move forward, left, right, then backward for testing and tuning.

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12

  forward(2000);                        // Go forward for 2 seconds
  turnLeft(600);                        // Turn left for 0.6 seconds
  turnRight(600);                       // Turn right for 0.6 seconds
  backward(2000);                       // go backward for 2 seconds

  disableServos();                      // Stay still indefinitely
}

void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
```

```
}

void forward(int time)                  // Forward function
{
  servoLeft.writeMicroseconds(1700);    // Left wheel counterclockwise
  servoRight.writeMicroseconds(1300);   // Right wheel clockwise
  delay(time);                          // Maneuver for time ms
}

void turnLeft(int time)                 // Left turn function
{
  servoLeft.writeMicroseconds(1300);    // Left wheel clockwise
  servoRight.writeMicroseconds(1300);   // Right wheel clockwise
  delay(time);                          // Maneuver for time ms
}

void turnRight(int time)                // Right turn function
{
  servoLeft.writeMicroseconds(1700);    // Left wheel counterclockwise
  servoRight.writeMicroseconds(1700);   // Right wheel counterclockwise
  delay(time);                          // Maneuver for time ms
}

void backward(int time)                 // Backward function
{
  servoLeft.writeMicroseconds(1300);    // Left wheel clockwise
  servoRight.writeMicroseconds(1700);   // Right wheel counterclockwise
  delay(time);                          // Maneuver for time ms
}

void disableServos()                    // Halt servo signals
{
  servoLeft.detach();                   // Stop sending servo signals
  servoRight.detach();
}
```

You should recognize the pattern of movement your BOE Shield-Bot makes; it is the same one made by the ForwardLeftRightBackward sketch. This is a second example of the many different ways to structure a sketch that will result in the same movements. There will be a few more examples before the end of the chapter.

Your Turn – Move Function Calls into loop

Want to keep performing that set of four maneuvers over and over again? Just move those four maneuvering function calls from the **`setup`** function into the **`loop`** function. Try this:

- ✓ Save the sketch under a new name, like MovementsWithFunctionsInLoop
- ✓ Comment out the **`disableServos()`** function call that's in **`setup`** by placing two forward slashes to its left, like this: **`// disable Servos`**
- ✓ Remove the **`// Empty…`** comment from the **`loop`** function—it won't be correct!

✓ Cut the function calls to **forward(2000)**, **turnLeft(600)**, **turnRight(600)**, and **backward(2000)** out of the **setup** function and paste them into the **loop** function. It should look like this when you're done:

```
void setup()                    // Built-in initialization block
{
  tone(4, 3000, 1000);          // Play tone for 1 second
  delay(1000);                  // Delay to finish tone

  servoLeft.attach(13);         // Attach left signal to pin 13
  servoRight.attach(12);        // Attach right signal to pin 12

  // disableServos();             // Stay still indefinitely
}

void loop()                     // Main loop auto-repeats
{
  forward(2000);                // Go forward for 2 seconds
  turnLeft(600);                // Turn left for 0.6 seconds
  turnRight(600);               // Turn right for 0.6 seconds
  backward(2000);               // Go backward for 2 seconds
```

✓ Run the modified sketch and verify that it repeats the sequence of four maneuvers indefinitely.

Activity 6: Custom Maneuver Function

The last sketch, MovementsWithSimpleFunctions, was kind of long and clunky. And, the four functions it uses to drive the robot are almost the same. The TestManeuverFunction sketch takes advantage of those function's similarities and streamlines the code.

TestManeuverFunction has a single function for motion named **maneuver** that accepts three parameters: **speedLeft**, **speedRight**, and **msTime**:

```
void maneuver(int speedLeft, int speedRight, int msTime)
```

The rules for **speedLeft** and **speedRight** are easy to remember. With this **maneuver** function you don't have to think about clockwise and counterclockwise rotation anymore.

- positive values for moving the robot forward
- negative values for moving the robot backward
- 200 for full speed forward
- –200 for full speed backward
- 0 for stop
- 100 to –100 range for linear speed control

The rules for **msTime** are:

- Positive values for the number of ms to execute the maneuver
- –1 to disable the servo signal

Here is what calls to this function will look like for the familiar forward-backward-left-right-stop sequence:

```
maneuver(200, 200, 2000);           // Forward 2 seconds
maneuver(-200, 200, 600);           // Left 0.6 seconds
maneuver(200, -200, 600);           // Right 0.6 seconds
maneuver(-200, -200, 2000);         // Backward 2 seconds
maneuver(0, 0, -1);                 // Disable servos
```

Example Sketch - TestManeuverFunction

- ✓ Create, save, and run TestManeuverFunction.
- ✓ Verify that it completes the forward, left, right, backward, stop sequence.

```
// Robotics with the BOE Shield - TestManeuverFunction
// Move forward, left, right, then backward with maneuver function.

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12

  maneuver(200, 200, 2000);             // Forward 2 seconds
  maneuver(-200, 200, 600);             // Left 0.6 seconds
  maneuver(200, -200, 600);             // Right 0.6 seconds
  maneuver(-200, -200, 2000);           // Backward 2 seconds
  maneuver(0, 0, -1);                   // Disable servos
}

void loop()                             // Main loop auto-repeats
{                                       // Empty, nothing needs repeating
}

void maneuver(int speedLeft, int speedRight, int msTime)
{
  // speedLeft, speedRight ranges: Backward Linear Stop Linear Forward
  //                                -200    -100...0....100     200
  servoLeft.writeMicroseconds(1500 + speedLeft);   // Left servo speed
  servoRight.writeMicroseconds(1500 - speedRight); // Right servo speed
  if(msTime==-1)                                   // if msTime = -1
```

```
  {
    servoLeft.detach();                          // Stop servo signals
    servoRight.detach();
  }
  delay(msTime);                                 // Delay for msTime
}
```

Your Turn – Customize Speed and Duration Control

With the **maneuver** function, 0 is stop, 100 to –100 is the speed control range, and 200 and -200 are overkill to keep the servos running as fast as they possibly can.

The TestManeuverFunction sketch makes it easy to define custom maneuvers quickly. Just pass new parameters for each wheel rotation and maneuver duration to each call of the **maneuver** function. For example, let's make the left wheel move at half speed while the right wheel moves at full speed to draw an arc for 3 seconds. Here is what that function call would look like:

```
maneuver(50, 100, 3000);
```

Here is another example that keeps the left wheel still and moves the right wheel forward for a left pivot turn:

```
maneuver(0, 200, 1200);
```

- ✓ Try adding both of the example **maneuver** calls to your sketch.
- ✓ Try adding the other three wheel-pivot turns to the sequence: forward-right, backward-right, and backward-left.

Activity 7: Maneuver Sequences with Arrays

Some robotics applications require sequences of maneuvers. You'll actually see some simple sequence examples in the next chapter when the BOE Shield-Bot detects that it has bumped into an obstacle. At that point, it has to back up, and then turn before trying to go forward again. That is a simple sequence with two maneuvers.

Another example is corridor navigation. The BOE Shield-Bot might have to find a corridor and then go through a sequence of maneuvers to enter it, before searching for the corridor's walls.

Other sequences can be more elaborate. One example of a long and complex maneuver would be for a robotic dance contest. (Robot dance contests have been gaining popularity in recent years.) For dancing to an entire song, the robot might need a pretty long list of

maneuvers and maneuver times. If your sketch needs to store lists of maneuvers, the variable array is the best tool for storing these lists.

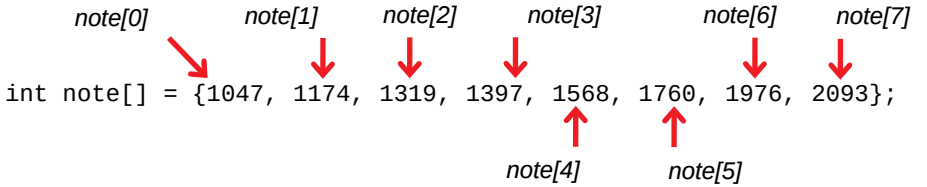
This activity introduces arrays with some simple musical applications using the piezospeaker. Then, it examines two different approaches for using arrays to store sequences of maneuvers for playback while the sketch is running.

What's an Array?

An *array* is a collection of variables with a common name. Each variable in the array is referred to as an *element*. Here is an array declaration with eight elements:

```
int note[] = {1047, 1174, 1319, 1397, 1568, 1760, 1976, 2093};
```

The array's name is **note**, and each element in the array can be accessed by its index number. Arrays are *zero indexed*, so the elements are numbered 0 to 7. The diagram below shows how **note[0]** stores 1047, **note[1]** stores 1174, **note[2]** stores 1319, and so on, up through **note[7]**, which stores 2093.



Let's say your code needs to copy **note[3]**, which stores the value 1397, to a variable named **myVar**. Your code could do it like this:

```
myVar = note[3];
```

Your sketch can change array element values too. For example, you could change the value 1976 to 1975 with an expression like this:

```
note[3] = 1975;
```

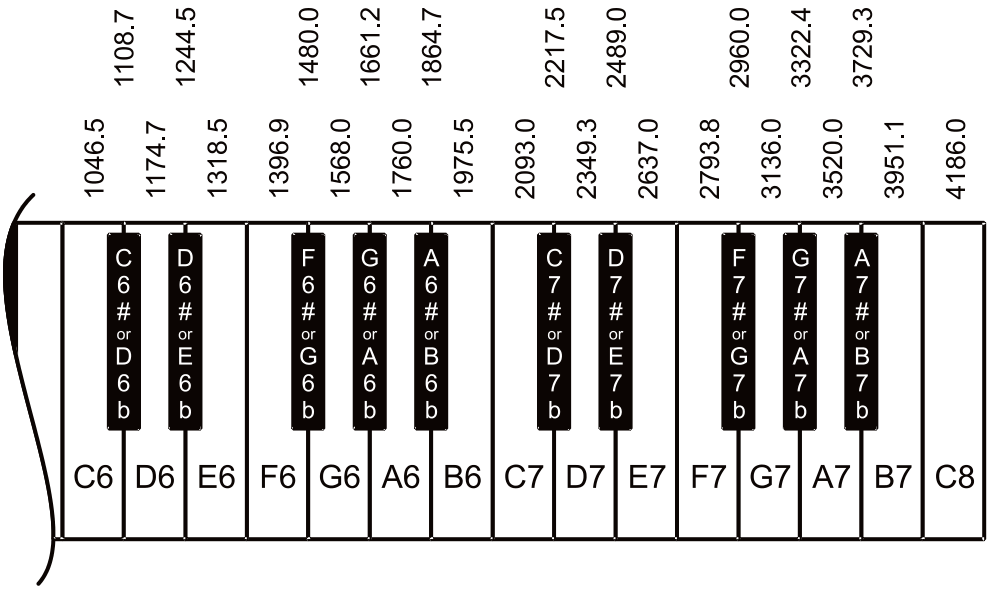
An array does not have to be pre-initialized with values like it is in the diagram above. For example, you could just declare an array with 8 elements like this:

```
int myArray[8];
```

Then, your sketch could fill in the values of each element later, perhaps with sensor measurements, values entered from the Serial Monitor, or whatever numbers you need to store.

The diagram below shows the musical notes on the right side of a piano keyboard. Each key press on a piano key makes a string (or a speaker if it's electric) vibrate at a certain frequency.

Compare the frequencies of the leftmost eight white keys to the values in the note array.



Using Array Elements

An array element doesn't necessarily need to be copied to another variable to use its value. For example, your sketch could just print the value in **`note[3]`** to the Serial Monitor like this:

```
Serial.print(note[3]);
```

Since the values in the array are musical notes, we might as well play this note on the BOE Shield-Bot's piezospeaker! Here's how:

```
tone(4, note[3], 500);
```

Example Sketch – PlayOneNote

Here is an example that displays an individual array element's value in the Serial Monitor, and also uses that value to make the BOE Shield-Bot's piezospeaker play a musical note.

- ✓ Create, save, and run the PlayOneNote sketch, and run it on the Arduino.
- ✓ Open the Serial Monitor as soon as the sketch is done loading.
- ✓ Verify that the Serial Monitor displays "note = 1397."
- ✓ Verify that the speaker played a tone.
- ✓ Modify the sketch to play and print the value of 1568 using **`note[4]`**.
- ✓ Test your modified sketch.

```
// Robotics with the BOE Shield – PlayOneNote
// Displays and plays one element from note array.

int note[] = {1047, 1147, 1319, 1397, 1568, 1760, 1976, 2093};

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

  Serial.print("note = ");
  Serial.println(note[3]);

  tone(4, note[3], 500);
  delay(750);
}

void loop()
{
}
```

Example Sketch – PlayNotesWithLoop

Many applications use variables to access elements in an array. The next sketch declares a variable named **`index`** and uses it to select an array element by its index number.

The familiar **`for`** loop can automatically increment the value of **`index`**. The code to play and display notes is inside the **`for`** loop, and **`index`** is used to select the array element. For the first trip through the loop, **`index`** will be 0, so the value stored in **`note[0]`** will be used wherever **`note[index]`** appears in a **`print`** or **`tone`** function. With each trip through the loop, **`index`** will increment until the sketch has displayed and played all the notes in the array.

- ✓ Create, save, and run PlayNotesWithLoop, then open the Serial Monitor as soon as the sketch is done loading.

✓ Verify that the Serial Monitor displays each note in the array as the speaker plays it.

```
// Robotics with the BOE Shield – PlayNotesWithLoop
// Displays and plays another element from note array.

int note[] = {1047, 1147, 1319, 1397, 1568, 1760, 1976, 2093};

void setup()
{

  Serial.begin(9600);

  for(int index = 0; index < 8; index++)
  {
    Serial.print("index = ");
    Serial.println(index);

    Serial.print("note[index] = ");
    Serial.println(note[index]);

    tone(4, note[index], 500);
    delay(750);
  }
}

void loop()
{
}
```

✓ What do you think will happen if you change the **for** loop to match the one below? Try it!

```
for(int index = 7; index >= 0; index--);
```

Using the sizeof Function

Let's say you want to compose a musical melody that has more, or fewer, notes. It's easy to forget to update the **for** loop to play the correct number of notes. The Arduino library has a **sizeof** function that can help with this. It can tell you both the size of the array in bytes, and the size of the array's variable type (like **int**). Your code can then divide the number of bytes for the variable type into the number of bytes in the array. The result is the number of elements in the array.

Here is an example of using this technique. It loads a variable named **elementCount** with the number of elements in the **note** array:

```
int note[] = {1047, 1147, 1319, 1397, 1568, 1760, 1976, 2093};
```

```
int elementCount = sizeof(note) / sizeof(int);
```

Later, your **`for`** loop can use the **`elementCount`** variable to play all the notes in the array, even if you add or delete elements:

```
  for(int index = 0; index < elementCount; index++)
```

- ✓ Create, save, and run PlayAllNotesInArray.
- ✓ Open the Serial Monitor as soon as the sketch is done loading.
- ✓ Verify again that the Serial Monitor displays each note in the array as the speaker plays it.

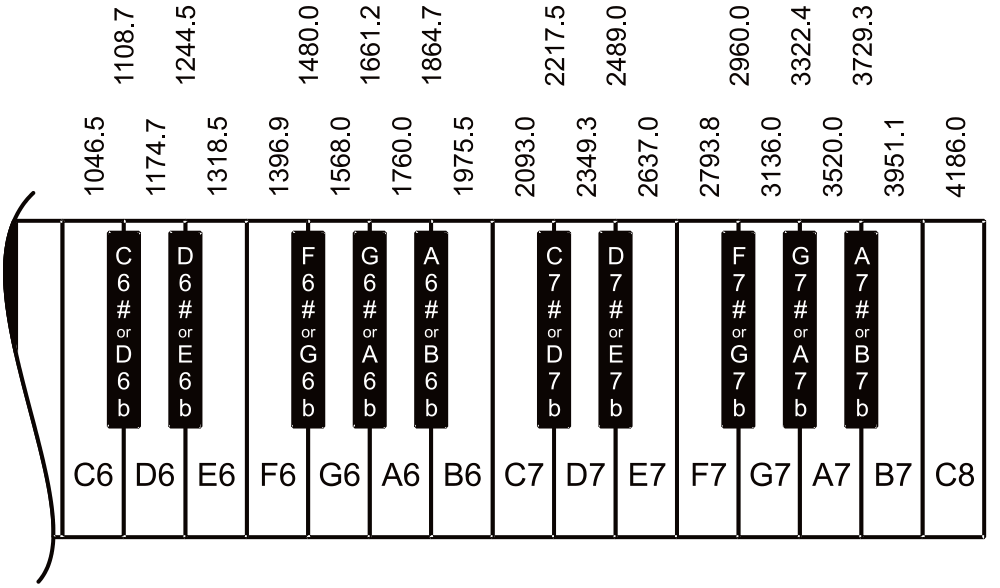
```
// Robotics with the BOE Shield – PlayAllNotesInArray
// Uses sizeof to determine number of elements in the array
// and then displays and prints each note value in the sequence.

int note[] = {1047, 1147, 1319, 1397, 1568, 1760, 1976, 2093};

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

  int elementCount = sizeof(note) / sizeof(int);

  Serial.print("Number of elements in array = ");
  Serial.println(elementCount);

  for(int index = 0; index < elementCount; index++)
  {
    Serial.print("index = ");
    Serial.println(index);

    Serial.print("note[index] = ");
    Serial.println(note[index]);

    tone(4, note[index], 500);
    delay(750);
  }
}

void loop()
{
}
```

Your Turn – Add Notes

- ✓ Try adding the next two notes, D7 and E7, using the frequencies (rounded down) from the keyboard diagram. Your array can use more than one line, like this:

```
int note[] = {1047, 1147, 1319, 1397, 1568, 1760,
              1976, 2093, 2349, 2637};
```

✓ If you are musically inclined, try writing an array that will play a very short tune.



Navigation with Arrays

Remember the maneuver function from the last activity? Here are three arrays of values, one for each parameter in the **maneuver** function. Together, they make up the same forward-left-right-backward-stop sequence we've been doing through the chapter.

```
//                Forward    left   right   backward    stop
//         index        0       1       2          3       4
int speedsLeft[]  = {200,    -200,    200,      -200,      0};
int speedsRight[] = {200,     200,   -200,      -200,      0};
int times[]       = {2000,    600,    600,      2000,     -1};
```

A sketch can then use this code in one of its functions to execute all the maneuvers:

```
// Determine number of elements in sequence list.
int elementCount = sizeof(times) / sizeof(int);

// Fetch successive elements from each sequence list and feed them
// to maneuver function.
```

```
  for(int index = 0; index < elementCount; index++)
  {
    maneuver(speedsLeft[index], speedsRight[index], times[index]);
  }
```

Each time through the loop, **index** increases by 1. So, with each **maneuver** call, the next element in each array gets passed as a parameter to the **maneuver** function. The first time through the loop, **index** is 0, so the **maneuver** call's parameters become the zeroth element from each array, like this: **maneuver(speedsLeft[0], speedsRight[0], times[0])**. The actual values that get passed to the **maneuver** function look like this: **maneuver(200, 200, 2000)**. The second time through the loop, **index** is 1, so the function looks like this: **maneuver(speedsLeft[1], speedsRight[1], times[1])**, which becomes **maneuver(-200, 200, 2000)**.

Example Sketch – ManeuverSequence

- ✓ Create, save, and run the ManeuverSequence sketch.
- ✓ Verify that the robot makes the forward-left-right-backward motions, then stops.

```
// Robotics with the BOE Shield - ManeuverSequence
// Move forward, left, right, then backward with an array and the
// maneuver function.

#include <Servo.h>                      // Include servo library

//                Forward    left   right   backward    stop
//         index        0       1       2          3       4
int speedsLeft[]  = {200,    -200,    200,      -200,      0};
int speedsRight[] = {200,     200,   -200,      -200,      0};
int times[]       = {2000,    600,    600,      2000,     -1};

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12
  // Determine number of elements in sequence list.
  int elementCount = sizeof(times) / sizeof(int);

  // Fetch successive elements from each sequence list and feed them
  // to maneuver function.
  for(int index = 0; index < elementCount; index++)
  {
    maneuver(speedsLeft[index], speedsRight[index], times[index]);
  }
```

```
}

void loop()                                // Main loop auto-repeats
{                                          // Empty, nothing needs repeating
}

void maneuver(int speedLeft, int speedRight, int msTime)
{
  // speedLeft, speedRight ranges: Backward  Linear  Stop  Linear   Forward
  //                               -200      -100......0......100       200
  servoLeft.writeMicroseconds(1500 + speedLeft);   // Left servo speed
  servoRight.writeMicroseconds(1500 - speedRight); // right servo speed
  if(msTime==-1)                                   // if msTime = -1
  {
    servoLeft.detach();                      // Stop servo signals
    servoRight.detach();
  }
  delay(msTime);                             // Delay for msTime
}
```

Did your BOE Shield-Bot perform the familiar forward-left-right-backward-stop sequence of movements? Are you thoroughly bored with it by now? Do you want to see your BOE Shield-Bot do something else, or to choreograph your own routine?

Your Turn – Add Maneuvers to the List

Here's an example of a longer list you can try. It does the four pivots after the forward-left-right-backward sequence. In this example, when **index** is 4, it'll use the first number of the second line of each array. When **index** is 5, it'll use the second number on the second line of each array, and so on. Notice that each list of comma-separated array elements is contained within curly braces **{ }**, and it doesn't matter whether that list is all on one line or spanning multiple lines.

```
int speedsLeft[]  = {200,    -200,    200,      -200,
                       0,     200,   -200,        0,       0};
int speedsRight[] = {200,     200,   -200,      -200,
                     200,       0,      0,      -200,      0};
int times[]       = {2000,   600,    600,      2000,
                     1000,   1000,   1000,     1000,    -1};
```

- ✓ Save ManeuverSequence as ManeuverSequenceExpanded.
- ✓ Change the array so it matches the examples with 9 elements above.
- ✓ Run the modified sketch and verify that the pivot sequence gets executed after the forward-left-right-backward sequence.

Character Arrays and switch-case

The last example in this activity is a sketch for performing maneuvers using a list of characters in an array. Each character represents a certain maneuver, with a 200 ms run time. Since the run time is fixed, it's not as flexible as the last approach, but it sure makes it simple to build a quick sequence of maneuvers.

f = forward **b** = backward **l** = left **r** = right **s** = stop

Character arrays do not use lists of comma-separated elements. Instead, they use a continuous string of characters. Here is an example of the same-old forward-left-right-backward-stop sequence in a character array:

```
char maneuvers[] = "ffffffffffllrrrbbbbbbbbbs";
```

The character array string has 10 **f** characters. Since each character represents 200 ms of run time, that takes the BOE Shield-Bot forward for 2 seconds. Next, three **l** characters make 600 ms of left turn. Three **r** characters make a right turn, followed by ten **b** characters to go backward, and then an **s** character for stop completes the sequence.

Example Sketch: ControlWithCharacters

- ✓ Create, save, and run ControlWithCharacters on the Arduino.
- ✓ Verify that the BOE Shield-Bot executes the forward-left-right-backward motions and then stops.

```
// Robotics with the BOE Shield – ControlWithCharacters
// Move forward, left, right, then backward for testing and tuning.

#include <Servo.h>                      // Include servo library

char maneuvers[] = "ffffffffffllrrrbbbbbbbbbs";

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to P13
  servoRight.attach(12);                // Attach right signal to P12

  // Parse maneuvers and feed each successive character to the go function.
  int index = 0;
  do
  {
    go(maneuvers[index]);
```

```
  } while(maneuvers[index++] != 's');}

void loop()                              // Main loop auto-repeats
{                                        // Empty, nothing needs repeating
}

void go(char c)                          // go function
{
  switch(c)                              // Switch to code based on c
  {
    case 'f':                                 // c contains 'f'
      servoLeft.writeMicroseconds(1700);      // Full speed forward
      servoRight.writeMicroseconds(1300);
      break;
    case 'b':                                 // c contains 'b'
      servoLeft.writeMicroseconds(1300);      // Full speed backward
      servoRight.writeMicroseconds(1700);
      break;
    case 'l':                                 // c contains 'l'
      servoLeft.writeMicroseconds(1300);      // Rotate left in place
      servoRight.writeMicroseconds(1300);
      break;
    case 'r':                                 // c contains 'r'
      servoLeft.writeMicroseconds(1700);      // Rotate right in place
      servoRight.writeMicroseconds(1700);
      break;
    case 's':                                 // c contains 's'
      servoLeft.writeMicroseconds(1500);      // Stop
      servoRight.writeMicroseconds(1500);
      break;
  }
  delay(200);                             // Execute for 0.2 seconds
}
```

✓ Try this array—can you guess what it will make the BOE Shield-Bot do?

```
char maneuvers[] = "fffffffffllllrrrrrrllllbbbbbbbbbbs";
```

After the **`char maneuvers`** array and the usual initialization, these lines fetch the characters from the array and pass them to the **`go`** function (explained later).

```
  int index = 0;
  do
  {
    go(maneuvers[index]);
  } while(maneuvers[index++] != 's');
```

First, **`index`** is declared and initialized to zero, to be used in a **`do-while`** loop. Similar to a regular **`while`** loop, **`do-while`** repeatedly executes commands inside its code block while a condition is true, but the **`while`** part comes *after* its block so the block always executes at

least once. Each time through the loop, **go(maneuvers[index])** passes the character at **maneuvers[index]** to the **go** function. The **++** in **index++** adds one to the **index** variable for the next time through the loop—recall that this is the *post increment* operator. This continues **while(maneuvers[index] != 's')** which means "while the value fetched from the **maneuvers** array is not equal to 's' ."

Now, let's look at the **go** function. It receives each character passed to its **c** parameter, and evaluates it on a case-by-case basis using a **switch/case** statement. For each of the five letters in the **maneuvers** character array, there is a corresponding **case** statement in the **switch(c)** block that will be executed if that character is received by **go**.

If the **go** function call passes the **f** character to the **c** parameter, the code in **case f** is executed—the familiar full-speed-forward. If it passes **b**, the full-speed backward code gets executed. The **break** in each case exits the **switch** block and the sketch moves on to the next command, which is **delay(200)**. So, each call to **go** results in a 200 ms maneuver. Without that break at the end of each case, the sketch would continue through to the code for the next case, resulting in un-requested maneuvers.

```
void go(char c)                                 // go function
{
  switch(c)                                     // Switch to based on c
  {
    case 'f':                                   // c contains 'f'
      servoLeft.writeMicroseconds(1700);        // Full speed forward
      servoRight.writeMicroseconds(1300);
      break;
    case 'b':                                   // c contains 'b'
      servoLeft.writeMicroseconds(1300);        // Full speed backward
      servoRight.writeMicroseconds(1700);
      break;
    case 'l':                                   // c contains 'l'
      servoLeft.writeMicroseconds(1300);        // Rotate left in place
      servoRight.writeMicroseconds(1300);
      break;
    case 'r':                                   // c contains 'r'
      servoLeft.writeMicroseconds(1700);        // Rotate right in place
      servoRight.writeMicroseconds(1700);
      break;
    case 's':                                   // c contains 's'
      servoLeft.writeMicroseconds(1500);        // Stop
      servoRight.writeMicroseconds(1500);
      break;
  }
  delay(200);                                    // Execute for 0.2 s
```

Your Turn – Add Array Elements and Cases

Try adding a **case** for half-speed forward. Use the character 'h', and remember that the linear speed range for the servos is from 1400 to 1600 microsecond pulses.

```
case 'h':                                 // c contains 'h'
  servoLeft.writeMicroseconds(1550);      // Half speed forward
  servoRight.writeMicroseconds(1450);
  break;
```

- ✓ Add ten or so **h** characters to your **maneuvers** character array. Keep in mind that they have to be added to the left of the **s** character for the sketch to get to them.
- ✓ Experiment a little, and add another **case** statement for a different maneuver, such as pivot-backward-left, then add some characters for the new maneuver to your array string. Can you see how this is a convenient way to build sequences of maneuvers?

Chapter 4 Summary

This chapter was all about robot navigation, experimenting with many different programming approaches and employing some robotics and engineering skills:

Programming

- Simplifying navigation by creating custom functions for frequently-used maneuver code, and how to call those functions
- How to use counted **for** loops with step increments in maneuver code
- What parameters are, and how to write functions and function calls that use them
- How a local variable can be created as a parameter in a function declaration
- How to declare, initialize, and use the Arduino language's **int** and **char** arrays, taking advantage of the Arduino library's **sizeof** function
- How to manage program flow control, using **do-while** and **switch/case**
- How to use the post increment operator (**++**) in conditional loops
- How to use the not-equal comparison operator (**!=**) as a condition in a loop
- Writing sketches to play a sequence of notes on the piezospeaker
- Writing sketches using several different program strategies to perform the same maneuver sequences

Robotics Skills

- How basic rolling-robot maneuvers are accomplished by controlling wheel speed and direction
- What the differences are between gradual turns, pivot-turns, and rotating-in-place turns, and what wheel speed/direction combinations make these turns.
- What speed ramping is, how to use it so your robot moves smoothly into and out of maneuvers, and how ramping is beneficial to your BOE Shield-Bot
- What dead reckoning is, in the context of entry-level robotics navigation

- Controlling robot maneuver run-time to make the BOE Shield-Bot travel a pre-determined distance or to rotate to a particular angle
- Compensating for hardware variance by adjusting servo speeds for straight travel

Engineering Skills

- Making observations and measurements to derive constants for a simple formula that characterizes the cause-and-effect relationship between system input and system output. (Yes it sounds fancy, but that's what you did with that ruler.)
- The difference between a hardware adjustment and a software adjustment
- What an iterative process is, and using it for testing software adjustments

Chapter 4 Challenges

Questions

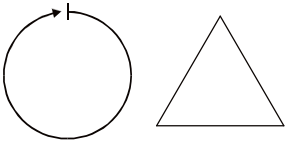
1. What direction does the left wheel have to turn to make the BOE Shield-Bot go forward? What direction does the right wheel have to turn?
2. When the BOE Shield-Bot pivots on one wheel to the left, what are the wheels doing? What code do you need to make the BOE Shield-Bot pivot left?
3. If your BOE Shield-Bot veers slightly to one side when you are running a sketch to make it go straight ahead, how do you correct this? What command needs to be adjusted and what kind of adjustment should you make?
4. If your BOE Shield-Bot travels 11 in/s, how many milliseconds will it take to make it travel 36 inches?
5. Why does a **`for`** loop that ramps servo speed need **`delay(20)`** in it?
6. What kind of variable is great for storing multiple values in lists?
7. What kind of loops can you use for retrieving values from lists?
8. What statement can you use to select a particular variable and evaluate it on a case-by-case basis and execute a different code block for each case?
9. What condition can you append to a **`do-loop`**?

Exercises

1. Write a routine that makes the BOE Shield-Bot back up at full speed for 2.5 seconds.
2. Let's say that you tested your servos and discovered that it takes 1.2 seconds to make a 180° turn with right-rotate. With this information, write routines to make the BOE Shield-Bot perform 30, 45, and 60 degree turns.
3. Write a routine that makes the BOE Shield-Bot go straight forward, then ramp into and out of a pivoting turn, and then continue straight forward.

Projects

1. It is time to fill in column 3 of the writeMicroseconds table on page 69. To do this, modify the ***us*** arguments in the **`writeMicroseconds`** calls in the ForwardThreeSeconds sketch using each pair of values from column 1. Record your BOE Shield-Bot's resultant behavior for each pair in column 3. Once completed, this table will serve as a reference guide when you design your own custom BOE Shield-Bot maneuvers.
2. The diagram shows two simple courses. Write a sketch that will make your BOE Shield-Bot navigate along each figure. Assume straight-line distances (the triangle's sides and the diameter of the circle) are either 1 yard or 1 meter.



Question Solutions

1. Left wheel counterclockwise, right wheel clockwise.
2. The right wheel is turning clockwise (forward), and the left wheel is not moving.

```
servoLeft.writeMicroseconds(1500);
servoRight.writeMicroseconds(1300);
```

3. Slow down the right wheel to correct a veer to the left, and slow down the left wheel to correct a veer to the right. Slow down a wheel by changing its servo's **`writeMicroseconds`** ***`us`*** parameter, using values closer to 1500. Start at the appropriate end of the linear speed control range (1400–1600), gradually move towards 1500 in increments of 10, and go back in smaller increments if you overshoot.

4. Given the data below, it should take about 3727 milliseconds to travel 36 inches:

   BOE Shield-Bot speed = 11 in/s

   BOE Shield-Bot distance = 36 in/s

   *time = (BOE Shield-Bot distance / BOE Shield-Bot speed) * (1000 ms / s)*

   *= (36 / 11) * (1000)*

   *= 3.727272…s * 1000 ms/s*

   ≈ 3727 ms

5. Without that 20 ms (1/50th of a second) delay between each repetition of the loop, it would ramp from 0 to 100 so quickly that it would seem like the BOE Shield-Bot just stepped instantly into full speed. The ramping would not be apparent.
6. An array.
7. **`for`** loops and **`do-while`** loops were examples from this chapter
8. **`switch/case`**.
9. **`do{...}loop while(condition)`**

Exercise Solutions

1. Solution:

```
servoLeft.writeMicroseconds(1300);
servoRight.writeMicroseconds(1700);
delay(2500);
```

2. Solution:

```
// 30/180 = 1/6, so use 1200/6 = 200
servoLeft.writeMicroseconds(1700);
servoRight.writeMicroseconds(1700);
delay(200);

// alternate approach
servoLeft.writeMicroseconds(1700);
servoRight.writeMicroseconds(1700);
delay(1200 * 30 / 180);

// 45/180 = 1/4, so use 1200/4 = 300
servoLeft.writeMicroseconds(1700);
```

```
servoRight.writeMicroseconds(1700);
delay(300);

// 60/180 = 1/3, so use 1200/3 = 400
servoLeft.writeMicroseconds(1700);
servoRight.writeMicroseconds(1700);
delay(400);
```

3. Solution:

```
// forward 1 second
servoLeft.writeMicroseconds(1700);
servoRight.writeMicroseconds(1700);
delay(1000);

// ramp into pivot
for(int speed = 0; speed <= 100; speed+=2)
{
  servoLeft.writeMicroseconds(1500);
  servoRight.writeMicroseconds(1500+speed);
  delay(20);
};

// ramp out of pivot
for(int speed = 100; speed >= 0; speed-=2)
{
  servoLeft.writeMicroseconds(1500);
  servoRight.writeMicroseconds(1500+speed);
  delay(20);
}

// forward again
servoLeft.writeMicroseconds(1700);
servoRight.writeMicroseconds(1700);
delay(1000);
```

Project Solutions

1. Solution (though the table looks a little different than the one you may have printed out.)

Servo ports connected to:		Description	Behavior
Pin 13	Pin 12		
1700	1300	Full Speed: pin 13 CCW, Pin 12 CW	Forward
1300	1700	Full Speed: Pin 13 CW, Pin 12 CCW	Backward
1700	1700	Full Speed: Pin 13 CCW, Pin 12 CCW	Right rotate
1300	1300	Full Speed: Pin 13 CW, Pin 12 CW	Left rotate
1500	1700	Pin 13 Stopped, Pin 12 CCW Full speed	Pivot back left
1300	750	Pin 13 CW Full Speed, Pin 12 Stopped	Pivot back right
1500	1500	Pin 13 Stopped, Pin 12 Stopped	Stopped
1520	1480	Pin 13 CCW Slow, Pin 12 CW Slow	Forward slow
1540	1460	Pin 13 CCW Med, Pin 12 CW Med	Forward medium
1700	1450	Pin 13 CCW Full Speed, Pin 12 CW Medium	Veer right
1550	1300	Pin 13 CCW Medium, Pin 12 CW Full Speed	Veer left

2. The circle can be implemented by veering right continuously. Trial and error, and a yard or meter stick, will help you arrive at the right ***us*** parameters for **`writeMicroseconds(`*`us`*`)`** and the right **`ms`** parameter for **`delay(`*`ms`*`)`**. Below is a solution that worked for a particular pair of servos and set of batteries. Your values may vary considerably from what's in the Circle sketch.

```
// Robotics with the BOE Shield - Chapter 4, project 2 - Circle
// BOE Shield-Bot navigates a circle of 1 yard diameter.

#include <Servo.h>                // Include servo library

Servo servoLeft;                  // Declare left and right servos
Servo servoRight;

void setup()                      // Built-in initialization block
```

```
{
  tone(4, 3000, 1000);           // Play tone for 1 second
  delay(1000);                   // Delay to finish tone

  servoLeft.attach(13);          // Attach left signal to pin 13
  servoRight.attach(12);         // Attach right signal to pin 12

  // Arc to the right
  servoLeft.writeMicroseconds(1600);  // Left wheel counterclockwise
  servoRight.writeMicroseconds(1438); // Right wheel clockwise slower
  delay(25500);                       // ...for 25.5 seconds

  servoLeft.detach();           // Stop sending servo signals
  servoRight.detach();
}

void loop()                     // Main loop auto-repeats
{                               // Nothing needs repeating
}
```

For the triangle, first calculate the required travel time in ms for a 1 meter or 1 yard straight line, as in Question 4, and fine-tune for your BOE Shield-Bot and particular surface. The BOE Shield-Bot must travel 1 meter/yard forward, and then make a 120° turn, repeated three times for the three sides of the triangle. You may have to adjust the delay call in the Turn Left 120° routine to get a precise 120° turn.

```
// Robotics with the BOE Shield - Chapter 4, project 2 - Triangle
// BOE Shield-Bot navigates a triangle with 1 yard sides and 120
// degree angles.  Go straight 1 yard, turn 120 degrees, repeat 3 times

#include <Servo.h>               // Include servo library

Servo servoLeft;                 // Declare left and right servos
Servo servoRight;

void setup()                     // Built-in initialization block
{
  tone(4, 3000, 1000);           // Play tone for 1 second
  delay(1000);                   // Delay to finish tone

  servoLeft.attach(13);          // Attach left signal to pin 13
  servoRight.attach(12);         // Attach right signal to pin 12

  for(int index = 1; index <= 3; index++)
  {
    // Full speed forward
    servoLeft.writeMicroseconds(1700);  // Left wheel counterclockwise
    servoRight.writeMicroseconds(1300); // Right wheel clockwise slower
    delay(5500);                        // ...for 5.5 seconds
```

```
    // Turn left 120 degrees
    servoLeft.writeMicroseconds(1300);  // Left wheel counterclockwise
    servoRight.writeMicroseconds(1300); // Right wheel clockwise slower
    delay(700);
  }
  servoLeft.detach();              // Stop sending servo signals
  servoRight.detach();
}

void loop()                        // Main loop auto-repeats
{                                  // Nothing needs repeating
}
```

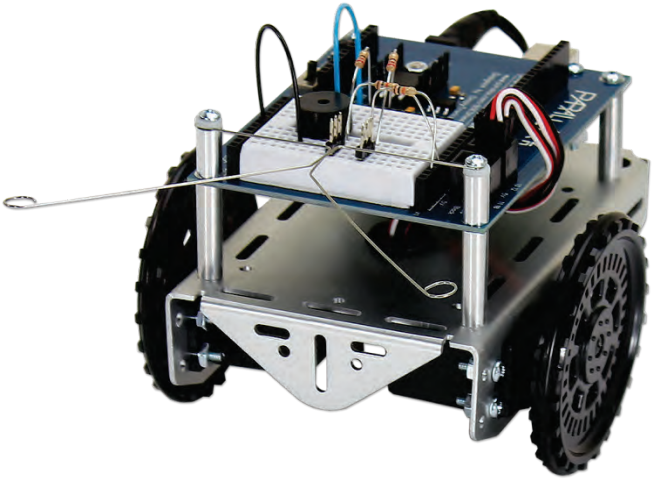
Chapter 5. Tactile Navigation with Whiskers

Tactile switches are also called bumper switches or touch switches, and they have many uses in robotics. A robot programmed to pick up an object and move it to another conveyer belt might rely on a tactile switch to detect the object. Automated factory lines might use tactile switches to count objects, and to align parts for a certain step in a manufacturing process. In each case, switches provide inputs that trigger some form of programmed output. The inputs are electronically monitored by the equipment's processor, which takes different actions depending on if the switch is pressed or not pressed.

In this chapter, you will build tactile switches, called *whiskers*, onto your BOE Shield-Bot and test them. You will then program the BOE Shield-Bot to monitor the states of these switches, and to decide what to do when it encounters an obstacle. The end result will be autonomous navigation by touch.

Tactile Navigation

Whisker switches give the BOE Shield-Bot the ability to sense its surroundings through touch as it roams around, much like a cat's whiskers. The activities in this chapter use the whiskers by themselves, but they can also be combined with other sensors you will learn about in later chapters.



Activity 1: Build and Test the Whiskers

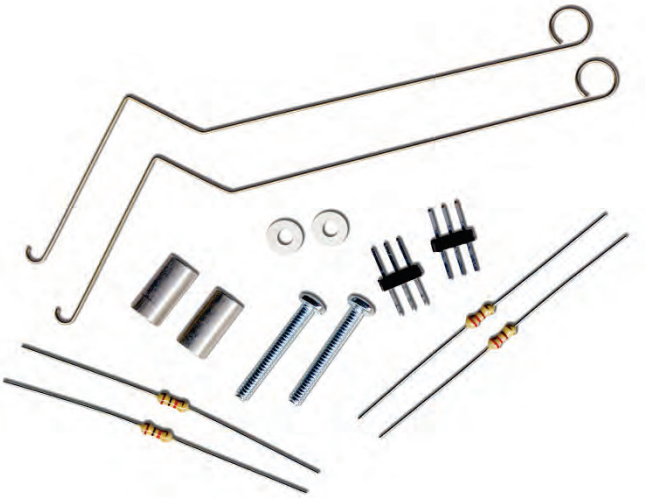
Remember subsystem testing? First, we'll build the whiskers circuits and write code to check their input states before using them in navigation sketches.

Whisker Circuit and Assembly

- ✓ Gather the whisker hardware in the parts list.
- ✓ Disconnect power from your board and servos.

Parts List

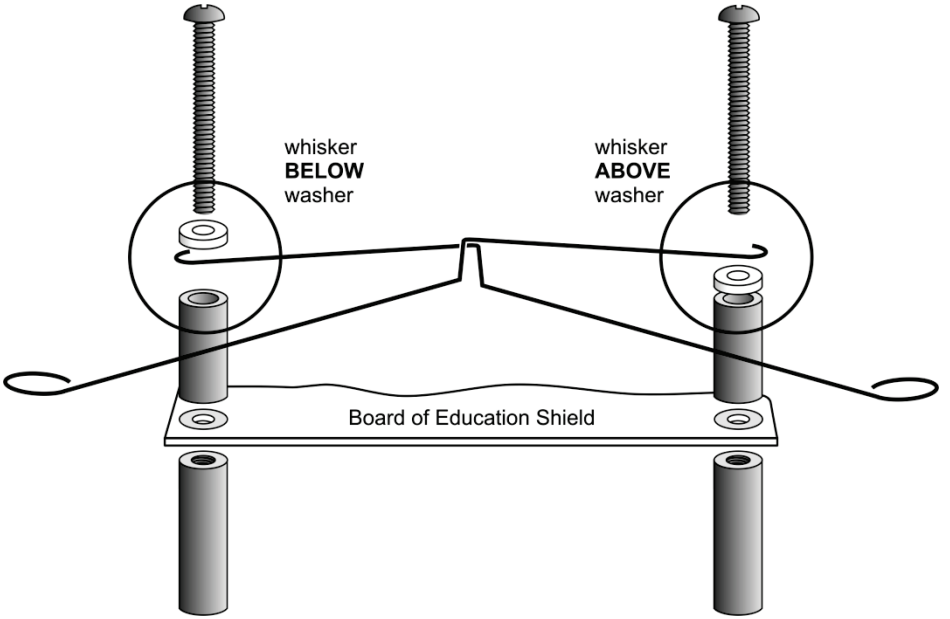
(2) whisker wires
(2) 7/8″ pan head 4-40 Phillips screws
(2) ½″ round spacer
(2) Nylon washers, size #4
(2) 3-pin m/m headers
(2) resistors, 220 Ω (red-red-brown)
(2) resistors, 10 kΩ (brown-black-orange)
(misc) jumper wires



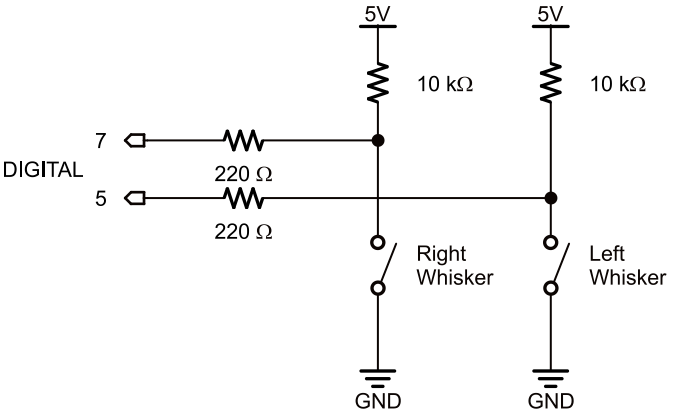
Building the Whiskers

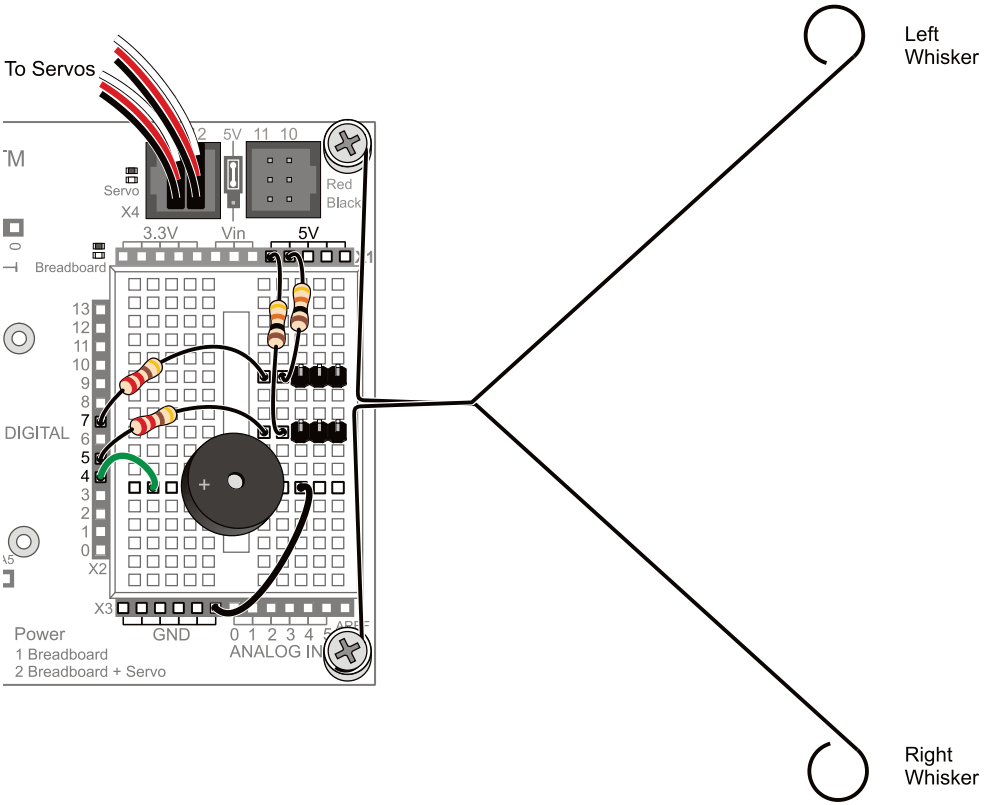
- ✓ Remove the LED circuits that were used as signal monitors while testing the servo navigation.
- ✓ Remove the two front screws that hold your board to the front standoffs.

- ✓ Thread a Nylon washer and then a ½″ round spacer on each of the 7/8″ screws.
- ✓ Attach the screws through the holes in your board and into the standoffs below, but do not tighten them all the way yet.
- ✓ Slip the hooked ends of the whisker wires around the screws, one above a washer and the other below a washer, positioning them so they cross over each other without touching.
- ✓ Tighten the screws into the standoffs.



- ✓ Use the 220 Ω resistors (red-red-brown) to connect digital pins 5 and 7 to their corresponding 3-pin headers.
- ✓ Use the 10 kΩ resistors (brown-black-orange) to connect 5 V to each 3-pin header.
- ✓ Make sure to adjust each whisker so that it is close to, but not touching, the 3-pin header on the breadboard. A distance of about 1/8″ (3 mm) is about right.


5V
5V
10 kΩ
10 kΩ
7
DIGITAL
220 Ω
5
220 Ω
Right
Whisker
Left
Whisker
GND
GND


To Servos
Left
Whisker
Servo
X4
Red
Black
3.3V
Vin
5V
Breadboard
DIGITAL
X2
X3
Power
1 Breadboard
2 Breadboard + Servo
GND
ANALOG IN
Right
Whisker

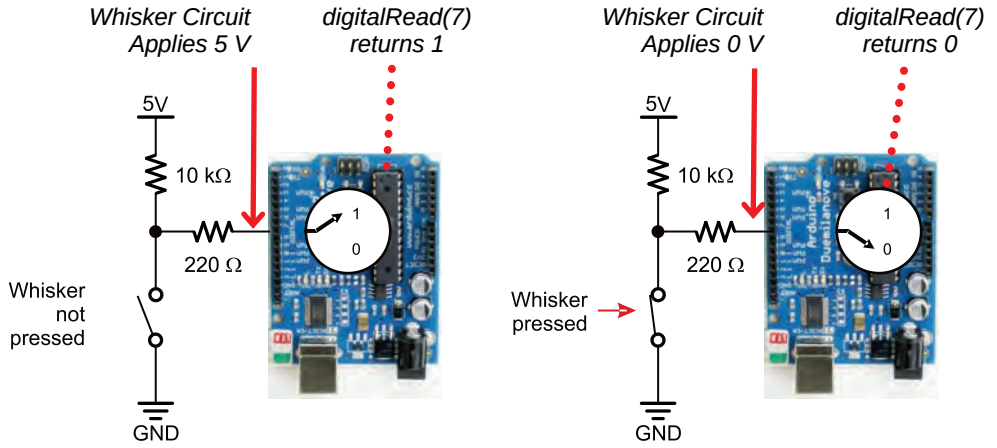
How Whisker Switches Work

The whiskers are connected to ground (GND) because the plated holes at the outer edge of the board are all connected to GND. The metal standoffs and screws provide the electrical connection to each whisker.

Since each whisker is connected to digital I/O, the Arduino can be programmed to detect which voltage is applied to each circuit, 5 V or 0 V. First, set each pin to input mode with **`pinMode(pin, mode)`**, and then detect the pin's state, **`HIGH`** or **`LOW`**, with the function **`digitalRead(pin)`**.

Take a look at the figure below. On the left, the circuit applies 5 V when the whisker is not pressed, so **`digitalRead(7)`** returns 1 (**`HIGH`**). On the right, the circuit applies 0 V when the whisker is pressed, so **`digitalRead(7)`** returns 0 (**`LOW`**).

Most importantly, your sketch can store the return values in variables, such as **`wLeft`** and **`wRight`**, and then use them to trigger actions or make decisions. The next example sketch will demonstrate how.



Switch Lingo: Each whisker is both the mechanical extension and the ground electrical connection of a normally open (off until pressed) momentary (on only while pressed) single-pole (one set of electrical contact points), single-throw (only one position conducts) switch.

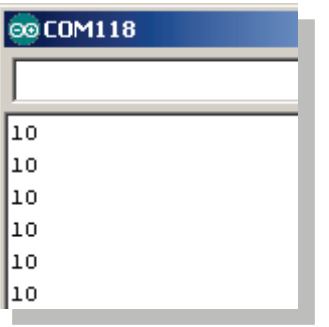
Testing the Whiskers

The next sketch tests the whiskers to make sure they are functioning properly, by displaying the binary values returned by **`digitalRead(7)`** and **`digitalRead(5)`**. This way, you can press each whisker against its 3-pin header on the breadboard, and see if the Arduino's digital pin is sensing the electrical contact.

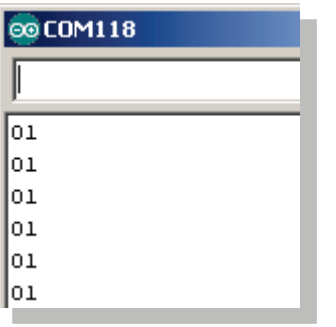
When neither whisker is pressed up against its 3-pin header, you can expect your Serial Monitor to display two columns of 1's, one for each whisker. If you press just the right whisker, the right column should report 0, and the display should read 10. If you press just the left whisker, the left column should report 1 and the display should read 01. Of course, if you press both whiskers, it should display 00.

No whiskers pressed

Right whisker pressed



Left whisker pressed



Active-low Output : The whisker circuits are wired for active-low output, which means that they each send a low signal when they are pressed (active) and a high signal when they are not pressed. Since **`digitalRead`** returns 0 for a low signal and 1 for a high signal, 0 is what tells your sketch that a whisker is pressed, and 1 tells it that a whisker is not pressed.

- ✓ Create, save, and run TestWhiskers on your Arduino.
- ✓ Reconnect the USB cable and set the 3-position switch to position 1.
- ✓ As soon as the sketch is finished loading, open the Serial Monitor.
- ✓ Leave the USB cable connected so that the Arduino can send serial messages to the Serial Monitor.

Example Sketch: DisplayWhiskerStates

```
/*
 * Robotics with the BOE Shield - DisplayWhiskerStates
 * Display left and right whisker states in Serial Monitor.
 * 1 indicates no contact; 0 indicates contact.
 */

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone
  pinMode(7, INPUT);                    // Set right whisker pin to input
  pinMode(5, INPUT);                    // Set left whisker pin to input
```

```
  Serial.begin(9600);                   // Set data rate to 9600 bps
}

void loop()                             // Main loop auto-repeats
{
  byte wLeft = digitalRead(5);          // Copy left result to wLeft
  byte wRight = digitalRead(7);         // Copy right result to wRight

  Serial.print(wLeft);                  // Display left whisker state
  Serial.println(wRight);               // Display right whisker state

  delay(50);                            // Pause for 50 ms
}
```

- ✓ Look at the values displayed in the Serial Monitor. With no whiskers pressed, it should display 11, indicating 5 V is applied to both digital inputs (5 and 7).
- ✓ Press the right whisker into its three-pin header, and note the values displayed in the Serial Monitor. It should now read 10.
- ✓ Release the right whisker and press the left whisker into its three-pin header, and note the value displayed in the Serial Monitor again. This time it should read 01.
- ✓ Press both whiskers against both three-pin headers. Now it should read 00.
- ✓ If the whiskers passed all these tests, you're ready to move on. If not, check your sketch and circuits for errors.

These steps are important! Seriously, you've got to make sure your circuit and code pass these tests before continuing. The rest of the examples in this chapter rely on the whiskers working correctly. If you haven't tested and corrected any errors, the rest of the examples won't work.

How DisplayWhiskerStates Works

In the **`setup`** function, **`pinMode(7, INPUT)`** and **`pinMode(5, INPUT)`** set digital pins 7 and 5 to input so they can monitor the voltages applied by the whisker circuits.

```
  pinMode(7, INPUT);          // Set right whisker pin to input
  pinMode(5, INPUT);          // Set left whisker pin to input
```

In the **`loop`** function, each call to **`digitalRead`** returns a 0 if the whisker is pressed or 1 if it is not. Those values get copied to variables named **`wLeft`** and **`wRight`**, which are short for whisker-left and whisker-right.

```
  byte wLeft = digitalRead(5);     // Copy left result to wLeft
  byte wRight = digitalRead(7);    // Copy right result to wRight
```

Next, **`Serial.print`** displays the value of **`wLeft`** to the Serial Monitor, and **`Serial.println`** displays the value of **`wRight`** and a carriage return.

```
  Serial.print(wLeft);              // Display left whisker state
  Serial.println(wRight);           // Display right whisker state
```

Before the next repetition of the **`loop`** function, there's a **`delay(50)`**. This slows down the number of messages the Serial Monitor receives each second. Although it's probably not needed, we leave it in to prevent possible *computer buffer overruns* (too much data to store) for older hardware and certain operating systems.

Your Turn – Nesting Function Calls

Your sketch doesn't actually need to use variables to store the values from **`digitalRead`**. Instead, the (1 or 0) value that **`digitalRead`** returns can be used directly by nesting the function call inside **`Serial.print`** and sending its return value straight to the Serial Monitor. In that case, your **`loop`** function would look like this:

```
void loop()                         // Main loop auto-repeats
{
  Serial.print(digitalRead(5));     // Display wLeft
  Serial.println(digitalRead(7));   // Display wRight

  delay(50);                        // Pause for 50 ms
}
```

- ✓ Replace the **`loop`** function with the one above, load the sketch, and test the whiskers to verify that it functions the same.

Activity 2: Field-Test the Whiskers

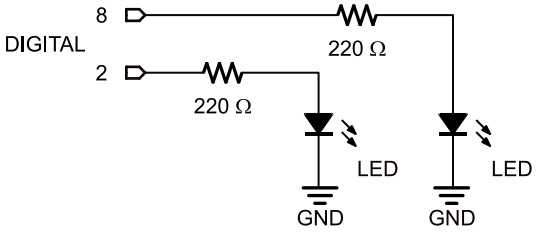
What if you have to test the whiskers at some later time away from a computer? In that case, the Serial Monitor won't be available, so what can you do? One solution would be to use LED circuits to display the whisker states. All it takes is a simple sketch that turns an LED on when a whisker is pressed or off when it's not pressed.

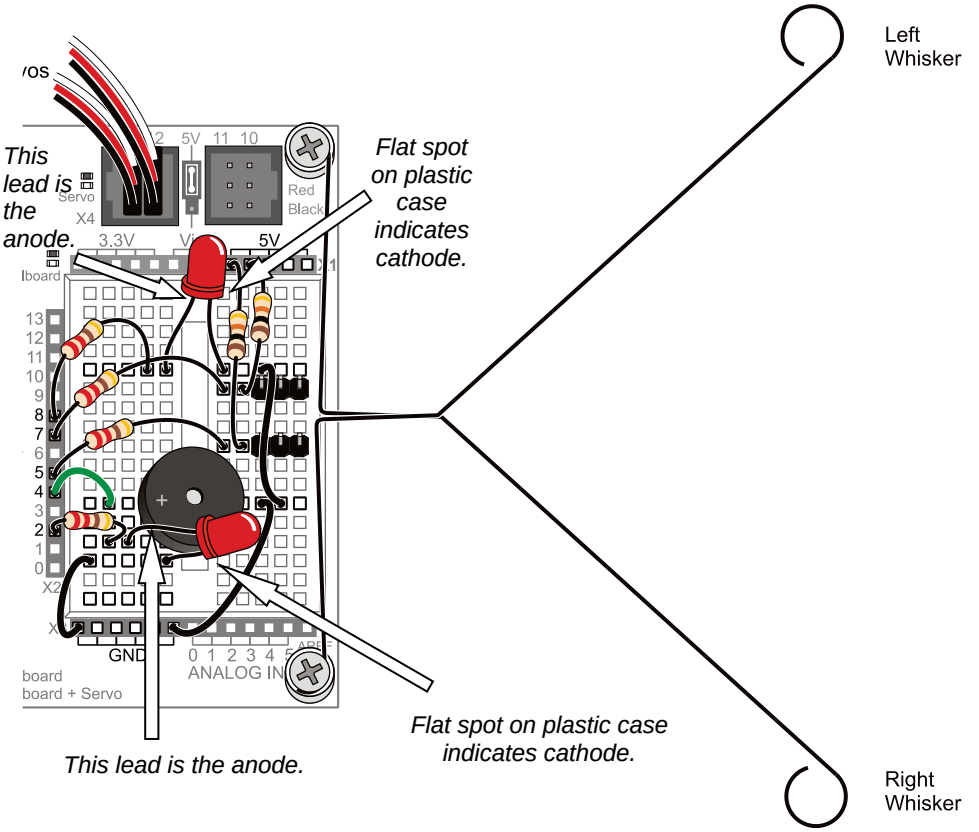
Parts List:

(2) resistors, 220 Ω (red-red-brown)
(2) LEDs, red

Build the LED Whisker Testing Circuits

- ✓ Unplug the BOE Shield-Bot's battery pack and USB cables.
- ✓ Add the circuit shown below.





Programming the LED Whisker Testing Circuits

- ✓ Re-save WhiskerStates as TestWhiskersWithLEDs.
- ✓ Add **`pinMode`** calls to the **`setup`** function, setting digital pins 8 and 2 to output.

```
pinMode(8, OUTPUT);         // Left LED indicator -> output
```

```
  pinMode(2, OUTPUT);         // Right LED indicator -> output
```

To make the whisker input states control the LEDs, insert these two **`if...else`** statements between the **`Serial.println(wRight)`** and **`delay(50)`** commands:

```
  if(wLeft == 0)              // If left whisker contact
  {
    digitalWrite(8, HIGH);    // Left LED on
  }
  else                        // If no left whisker contact
  {
    digitalWrite(8, LOW);     // Left LED off
  }

  if(wRight == 0)             // If right whisker contact
  {
    digitalWrite(2, HIGH);    // Right LED on
  }
  else                        // If no right whisker contact
  {
    digitalWrite(2, LOW);     // Right LED off
  }
```

Recall that **`if...else`** statements execute blocks of code based on conditions. Here, if **`wLeft`** stores a zero, it executes the **`digitalWrite(8, HIGH)`** call. If **`wLeft`** instead stores a 1, it executes the **`digitalWrite(8, LOW)`** call. The result? The left LED turns on when the left whisker is pressed or off when it's not pressed. The second **`if…else`** statement does the same job with **`wRight`** and the right LED circuit.

- ✓ Set the BOE Shield's power switch to position 1.
- ✓ Reconnect the Arduino's programming cable.
- ✓ Create, save, and run TestWhiskersWithLeds on your Arduino.
- ✓ Test the sketch by gently pressing each whisker against its 3-pin header post in the breadboard. The red LEDs on the side of the breadboard where you pressed the whisker should emit light to indicate that the whisker has made contact.
- ✓ If both LEDs light up and just stay on no matter what, your power switch is probably in position 0. Switch it to position 1 and try again.

```
/*
 * Robotics with the BOE Shield - TestWhiskersWithLeds
 * Display left and right whisker states in Serial Monitor.
 * 1 indicates no contact; 0 indicates contact.
 * Display whisker states with LEDs.  LED on indicates contact;
 * off indicates none.
 */
```

```
void setup()                          // Built-in initialization block
{
  pinMode(7, INPUT);                  // Set right whisker pin to input
  pinMode(5, INPUT);                  // Set left whisker pin to input
  pinMode(8, OUTPUT);                 // Left LED indicator -> output
  pinMode(2, OUTPUT);                 // Right LED indicator -> output

  tone(4, 3000, 1000);                // Play tone for 1 second
  delay(1000);                        // Delay to finish tone

  Serial.begin(9600);                 // Set serial data rate to 9600
}

void loop()                           // Main loop auto-repeats
{
  byte wLeft = digitalRead(5);        // Copy left result to wLeft
  byte wRight = digitalRead(7);       // Copy right result to wRight

  if(wLeft == 0)                      // If left whisker contact
  {
    digitalWrite(8, HIGH);            // Left LED on
  }
  else                                // If no left whisker contact
  {
    digitalWrite(8, LOW);             // Left LED off
  }

  if(wRight == 0)                     // If right whisker contact
  {
    digitalWrite(2, HIGH);            // Right LED on
  }
  else                                // If no right whisker contact
  {
    digitalWrite(2, LOW);             // Right LED off
  }

  Serial.print(wLeft);                // Display wLeft
  Serial.println(wRight);             // Display wRight
  delay(50);                          // Pause for 50 ms
}
```

Activity 3: Navigation with Whiskers

Previously, our sketches only made the BOE Shield-Bot execute a list of movements predefined by you, the programmer. Now that you can write a sketch to make the Arduino monitor whisker switches and trigger action in response, you can also write a sketch that lets the BOE Shield-Bot drive and select its own maneuver if it bumps into something. This is an example of *autonomous robot navigation*.

Whisker Navigation Overview

The RoamingWithWhiskers sketch makes the BOE Shield-Bot go forward while monitoring its whisker inputs, until it encounters an obstacle with one or both of them. As soon as the Arduino senses whisker electrical contact, it uses **`an if…else if…else`** statement to decide what to do. The decision code checks for various whisker pressed/not pressed combinations, and calls navigation functions to execute back-up-and-turn maneuvers. Then, the BOE Shield-Bot resumes forward motion until it bumps into another obstacle.

Example Sketch: RoamingWithWhiskers

Let's try the sketch first, and then take a closer look at how it works.

- ✓ Set the 3-position switch to position 1.
- ✓ Reconnect the BOE Shield-Bot's battery pack to the Arduino.
- ✓ Create, save, and run RoamingWithWhiskers.
- ✓ Disconnect the BOE Shield-Bot from its programming cable, and set the power switch to 2.
- ✓ Put the BOE Shield-Bot on the floor, and try letting it roam. When it contacts obstacles in its path with its whisker switches, it should back up, turn, and then roam in a new direction.

```
// Robotics with the BOE Shield - RoamingWithWhiskers
// Go forward.  Back up and turn if whiskers indicate BOE Shield-Bot bumped
// into something.

#include <Servo.h>                    // Include servo library

Servo servoLeft;                      // Declare left and right servos
Servo servoRight;

void setup()                          // Built-in initialization block
{
  pinMode(7, INPUT);                  // Set right whisker pin to input
  pinMode(5, INPUT);                  // Set left whisker pin to input

  tone(4, 3000, 1000);                // Play tone for 1 second
  delay(1000);                        // Delay to finish tone

  servoLeft.attach(13);               // Attach left signal to pin 13
  servoRight.attach(12);              // Attach right signal to pin 12
}

void loop()                           // Main loop auto-repeats
{
  byte wLeft = digitalRead(5);        // Copy left result to wLeft
  byte wRight = digitalRead(7);       // Copy right result to wRight

  if((wLeft == 0) && (wRight == 0))   // If both whiskers contact
```

```
  {
    backward(1000);                   // Back up 1 second
    turnLeft(800);                    // Turn left about 120 degrees
  }
  else if(wLeft == 0)                 // If only left whisker contact
  {
    backward(1000);                   // Back up 1 second
    turnRight(400);                   // Turn right about 60 degrees
  }
  else if(wRight == 0)                // If only right whisker contact
  {
    backward(1000);                   // Back up 1 second
    turnLeft(400);                    // Turn left about 60 degrees
  }
  else                                // Otherwise, no whisker contact
  {
    forward(20);                      // Forward 1/50 of a second
  }
}

void forward(int time)                // Forward function
{
  servoLeft.writeMicroseconds(1700);    // Left wheel counterclockwise
  servoRight.writeMicroseconds(1300);   // Right wheel clockwise
  delay(time);                          // Maneuver for time ms
}

void turnLeft(int time)                 // Left turn function
{
  servoLeft.writeMicroseconds(1300);    // Left wheel clockwise
  servoRight.writeMicroseconds(1300);   // Right wheel clockwise
  delay(time);                          // Maneuver for time ms
}

void turnRight(int time)                // Right turn function
{
  servoLeft.writeMicroseconds(1700);    // Left wheel counterclockwise
  servoRight.writeMicroseconds(1700);   // Right wheel counterclockwise
  delay(time);                          // Maneuver for time ms
}

void backward(int time)                 // Backward function
{
  servoLeft.writeMicroseconds(1300);    // Left wheel clockwise
  servoRight.writeMicroseconds(1700);   // Right wheel counterclockwise
  delay(time);                          // Maneuver for time ms
}
```

How RoamingWithWhiskers Works

The **`if...else if...else`** statement in the **`loop`** function checks the whiskers for any states that require attention. The statement starts with **`if((wLeft == 0) && (wRight`**

== 0)). Translated to English, it reads "if the **wLeft** variable AND the **wRight** variable both equal zero." If both variables are zero, the two calls in the **if** statement's code block get executed: **backward(1000)** and **turnLeft(800)**.

```
if((wLeft == 0) && (wRight == 0)) // If both whiskers contact
{
  backward(1000);                 // Back up 1 second
  turnLeft(800);                  // Turn left about 120 degrees
}
```

In the **if…else if…else** statement, the sketch skips code blocks with conditions that are not true, and keeps checking until it either finds a condition that's true or runs out of conditions. When the sketch finds a true statement, it executes whatever is in its code block, then it skips to the end of the **if…else if…else** statement without checking any more conditions, and moves on to whatever else comes next in the sketch.

So, if both whiskers are not pressed, that first **if** statement is not true and its code block is skipped. The sketch will check the first **else if** statement. So, maybe the left whisker is pressed and the calls in this statement's code block will run. After backing up for one second and turning right for 0.4 seconds, the sketch skips the rest of the conditions and moves on to whatever comes after that last **else** statement.

```
else if(wLeft == 0)               // If only left whisker contact
{
  backward(1000);                 // Back up 1 second
  turnRight(400);                 // Turn right about 60 degrees
}
```

If it's the right whisker that detects an obstacle, the first two code blocks will be skipped, and the **if(wRight == 0)** block will run.

```
else if(wRight == 0)              // If only right whisker contact
{
  backward(1000);                 // Back up 1 second
  turnLeft(400);                  // Turn left about 60 degrees
}
```

An **else** condition functions as a catch-all for when none of the statements preceding it were true. It's not required, but in this case, it's useful for when no whiskers are pressed. If that's the case, it allows the BOE Shield-Bot to roll forward for 20 ms. Why so little time before the loop repeats? The small forward time before rechecking allows the BOE Shield-Bot to respond quickly to changes in the whisker sensors as it rolls forward.

```
else                              // Otherwise, no whisker contact
{
```

```
  forward(20);                    // Forward 1/50 of a second
}
```

The **forward**, **backward**, **turnLeft** and **turnRight** functions were introduced in Chapter 4, Activity 5: Simplify Navigation with Functions on page 118, and are used in the MovementsWithSimpleFunctions sketch. These functions certainly simplified the coding. (Hopefully, they also help demonstrate that all the navigation coding practice from Chapter 4 has its uses!)

Your Turn

You can also modify the sketch's **if...else if...else** statements to make the LED indicators broadcast which maneuver the BOE Shield-Bot is running. Just add calls to **digitalWrite** that send **HIGH** and **LOW** signals to the indicator LED circuits. Here is an example:

```
if((wLeft == 0) && (wRight == 0)) // If both whiskers contact
{
  digitalWrite(8, HIGH);          // Left LED on
  digitalWrite(2, HIGH);          // Right LED on
  backward(1000);                 // Back up 1 second
  turnLeft(800);                  // Turn left about 120 degrees
}
else if(wLeft == 0)               // If only left whisker contact
{
  digitalWrite(8, HIGH);          // Left LED on
  digitalWrite(2, LOW);           // Right LED off
  backward(1000);                 // Back up 1 second
  turnRight(400);                 // Turn right about 60 degrees
}
else if(wRight == 0)              // If only right whisker contact
{
  digitalWrite(8, LOW);           // Left LED off
  digitalWrite(2, HIGH);          // Right LED on
  backward(1000);                 // Back up 1 second
  turnLeft(400);                  // Turn left about 60 degrees
}
else                              // Otherwise, no whisker contact
{
  digitalWrite(8, LOW);           // Left LED off
  digitalWrite(2, LOW);           // Right LED off
  forward(20);                    // Forward 1/50 of a second
}
```

- ✓ Modify the **if...else if...else** statement in RoamingWithWhiskers to make the BOE Shield-Bot broadcast its maneuver using the LED indicators.

- ✓ Remember to set the digital pins to outputs in the **`setup`** function so they can actually supply current to the LEDS:

```
pinMode(8, OUTPUT);          // Left LED indicator -> output
pinMode(2, OUTPUT);          // Right LED indicator -> output
```

Activity 4: Artificial Intelligence for Escaping Corners

You may have noticed that with the last sketch, the BOE Shield-Bot tends to get stuck in corners. As it enters a corner, its left whisker contacts the wall on the left, so it backs up and turns right. When the BOE Shield-Bot moves forward again, its right whisker contacts the wall on the right, so it backs up and turns left. Then it contacts the left wall again, and then the right wall again, and so on, until somebody rescues it from its predicament.

Programming to Escape Corners

RoamingWithWhiskers can be expanded to detect this problem and act upon it. The trick is to count the number of times that alternate whiskers make contact with objects. To do this, the sketch has to remember what state each whisker was in during the previous contact. Then, it has to compare those states to the current whisker contact states. If they are opposite, then add 1 to a counter. If the counter goes over a threshold that you (the programmer) have determined, then it's time to do a U-turn and escape the corner, and also reset the counter.

This next sketch relies on the fact that you can nest **`if`** statements, one inside another. The sketch checks for one condition, and if that condition is true, it checks for another condition within the first **`if`** statement's code block. We'll use this technique to detect consecutive alternate whisker contacts in the next sketch.

Example Sketch: EscapingCorners

This sketch will cause your BOE Shield-Bot to execute a reverse and U-turn to escape a corner at either the fourth or fifth alternate whisker press, depending on which one was pressed first.

- ✓ With the power switch in Position 1, create, save, and run EscapingCorners.
- ✓ Test this sketch pressing alternate whiskers as the BOE Shield-Bot roams. It should execute its reverse and U-turn maneuver after either the fourth or fifth consecutive, alternate whisker contact.

```
/*
 * Robotics with the BOE Shield - EscapingCorners
 * Count number of alternate whisker contacts; if it exceeds 4, get out
 * of the corner.
 */
```

```
#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

byte wLeftOld;                          // Previous loop whisker values
byte wRightOld;
byte counter;                           // For counting alternate corners

void setup()                            // Built-in initialization block
{
  pinMode(7, INPUT);                    // Set right whisker pin to input
  pinMode(5, INPUT);                    // Set left whisker pin to input
  pinMode(8, OUTPUT);                   // Left LED indicator -> output
  pinMode(2, OUTPUT);                   // Right LED indicator -> output

  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12

  wLeftOld = 0;                         // Init. previous whisker states
  wRightOld = 1;
  counter = 0;                          // Initialize counter to 0
}

void loop()                             // Main loop auto-repeats
{

  // Corner Escape
  byte wLeft = digitalRead(5);          // Copy right result to wLeft
  byte wRight = digitalRead(7);         // Copy left result to wRight

  if(wLeft != wRight)                   // One whisker pressed?
  {                                     // Alternate from last time?
    if ((wLeft != wLeftOld) && (wRight != wRightOld))
    {
      counter++;                        // Increase count by one
      wLeftOld = wLeft;                 // Record current for next rep
      wRightOld = wRight;
      if(counter == 4)                  // Stuck in a corner?
      {
        wLeft = 0;                      // Set up for U-turn
        wRight = 0;
        counter = 0;                    // Clear alternate corner count
      }
    }
    else                                // Not alternate from last time
    {
      counter = 0;                      // Clear alternate corner count
    }
  }
```

```
  // Whisker Navigation
  if((wLeft == 0) && (wRight == 0))     // If both whiskers contact
  {
    backward(1000);                     // Back up 1 second
    turnLeft(800);                      // Turn left about 120 degrees
  }
  else if(wLeft == 0)                   // If only left whisker contact
  {
    backward(1000);                     // Back up 1 second
    turnRight(400);                     // Turn right about 60 degrees
  }
  else if(wRight == 0)                  // If only right whisker contact
  {
    backward(1000);                     // Back up 1 second
    turnLeft(400);                      // Turn left about 60 degrees
  }
  else                                  // Otherwise, no whisker contact
  {
    forward(20);                        // Forward 1/50 of a second
  }
}

void forward(int time)                  // Forward function
{
  servoLeft.writeMicroseconds(1700);    // Left wheel counterclockwise
  servoRight.writeMicroseconds(1300);   // Right wheel clockwise
  delay(time);                          // Maneuver for time ms
}

void turnLeft(int time)                 // Left turn function
{
  servoLeft.writeMicroseconds(1300);    // Left wheel clockwise
  servoRight.writeMicroseconds(1300);   // Right wheel clockwise
  delay(time);                          // Maneuver for time ms
}

void turnRight(int time)                // Right turn function
{
  servoLeft.writeMicroseconds(1700);    // Left wheel counterclockwise
  servoRight.writeMicroseconds(1700);   // Right wheel counterclockwise
  delay(time);                          // Maneuver for time ms
}

void backward(int time)                 // Backward function
{
  servoLeft.writeMicroseconds(1300);    // Left wheel clockwise
  servoRight.writeMicroseconds(1700);   // Right wheel counterclockwise
  delay(time);                          // Maneuver for time ms
}
```

How Escaping Corners Works

This sketch is a modified version of RoamingWithWhiskers, so we'll just look at the new code for detecting and escaping corners.

First, three global byte variables are added: **wLeftOld**, **wRightOld**, and **counter**. The **wLeftOld** and **wRightOld** variables store the whisker states from a previous whisker contact so that they can be compared with the states of the current contact. Then **counter** is used to track the number of consecutive, alternate contacts.

```
byte wLeftOld;                // Previous loop whisker values
byte wRightOld;
byte counter;                 // For counting alternate corners
```

These variables are initialized in the **setup** function. The **counter** variable can start with zero, but one of the "old" variables has to be set to 1. Since the routine for detecting corners always looks for an alternating pattern, and compares it to the previous alternating pattern, there has to be an initial alternate pattern to start with. So, **wLeftOld** and **wRightOld** are assigned initial values in the **setup** function before the **loop** function starts checking and modifying their values.

```
  wLeftOld = 0;               // Initialize previous whisker
  wRightOld = 1;              // states
  counter = 0;                // Initialize counter to 0
```

The first thing the code below **// Corner Escape** has to do is check if one or the other whisker is pressed. A simple way to do this is to use the not-equal operator (**!=**) in an **if** statement. In English, **if(wLeft != wRight)** means "if the **wLeft** variable is not equal to the **wRight** variable…"

```
  // Corner Escape

  if(wLeft != wRight)         // One whisker pressed?
```

If they are not equal it means one whisker is pressed, and the sketch has to check whether it's the opposite pattern as the previous whisker contact. To do that, a nested **if** statement checks if the current **wLeft** value is different from the previous one and if the current **wRight** value is different from the previous one. That's **if((wLeft != wLeftOld) && (wRight != wRightOld))**. If both conditions are true, it's time to add 1 to the **counter** variable that tracks alternate whisker contacts. It's also time to remember the current whisker pattern by setting **wLeftOld** equal to the current **wLeft** and **wRightOld** equal to the current **wRight**.

```
    if((wLeft != wLeftOld) && (wRight != wRightOld))
    {
      counter++;              // Increase count by one
      wLeftOld = wLeft;       // Record current for next rep
      wRightOld = wRight;
```

If this is the fourth consecutive alternate whisker contact, then it's time to reset the **counter** variable to 0 and execute a U-turn. When the **if(counter == 4)** statement is true, its code block tricks the whisker navigation routine into thinking both whiskers are pressed. How does it do that? It sets both **wLeft** and **wRight** to zero. This makes the whisker navigation routine think both whiskers are pressed, so it makes a U-turn.

```
      if(counter == 4)      // Stuck in a corner?
      {
        wLeft = 0;          // Set up whisker states for U-turn
        wRight = 0;
        counter = 0;        // Clear alternate corner count
      }
    }
```

But, if the conditions in **if((wLeft != wLeftOld) && (wRight != wRightOld))** are not all true, it means that this is not a sequence of alternating whisker contacts anymore, so the BOE Shield-Bot must not be stuck in a corner. In that case, the **counter** variable is set to zero so that it can start counting again when it really does find a corner.

```
    else                      // Not alternate from last time
    {
      counter = 0;            // Clear alternate corner count
    }
  }
```

One thing that can be tricky about nested **if** statements is keeping track of opening and closing braces for each statement's code block. The picture below shows some nested **if** statements from the last sketch. In the Arduino and codebender : edu editors, you can double-click on a brace to highlight its code block. But sometimes, printing out the code and simply drawing lines to connect opening and closing braces helps to see all the blocks at once, which is useful for finding bugs in deeply nested code.

In this picture, the **if(wLeft != wRight)** statement's code block contains all the rest of the decision-making code. If it turns out that **wLeft** is equal to **wRight**, the Arduino skips to whatever code follows that last closing brace **}**. The second level **if** statement compares the old and new **wLeft** and **wRight** values with **if ((wLeft != wLeftOld) && (wRight != wRightOld))**. Notice that its code block ending brace is just below the one for the **if(counter==4)** block. The **if ((wLeft != wLeftOld) && (wRight != wRightOld))**

statement also has an **else** condition with a block that sets **counter** to zero if the whisker values are not opposite from those of the previous contact.

```
// Corner Escape

byte wLeft = digitalRead(5);
byte wRight = digitalRead(7);

if(wLeft != wRight)
{
  if ((wLeft != wLeftOld) && (wRight != wRightOld))
  {
    counter++;
    wLeftOld = wLeft;
    wRightOld = wRight;
    if(counter == 4)
    {
      wLeft = 0;
      wRight = 0;
      counter = 0;
    }
  }
  else
  {
    counter = 0;
  }
}
```

- ✓ Study the code in the picture carefully.
- ✓ Imagine that **wLeft** = 0 , **wRight** = 0 and **counter** == 3, and think about what this statement would do.
- ✓ Imagine that **wLeft** = 1 , **wRight** = 0 , **wLeftOld** = 0 , **wRight** = 1 and **counter** == 3. Try walking through the code again line by line and explain what happens to each variable at each step.

Your Turn

One of the **if** statements in EscapingCorners checks to see if **counter** has reached 4.

- ✓ Try increasing the value to 5 and 6 and test the effect. Keep in mind that it will either count to the number of alternate whisker contacts, or maybe one more than that depending on which side you start.

Chapter 5 Summary

This chapter introduced the first sensor system for the BOE Shield-Bot, and allowed the robot to roam around on its own and navigate by touch. The project built on skills acquired in the last chapter, and employed a variety of new ones:

Electronics

- Building normally open, momentary, single-pole, single-throw, tactile switch circuits

Programming

- Using the Arduino language's **`pinMode`** and **`digitalRead`** functions to set a digital I/O pin to input and then monitor the pin's input state
- Declaring and initializing global variables
- Nesting function calls
- Using **`if…else`** statements to direct program flow based on sensor input states
- Using three levels of nested **`if`** statements

Robotics Skills

- Autonomous robot navigation based on sensor inputs
- Using a simple example of artificial intelligence so the autonomously roaming BOE Shield-Bot can tell if it's stuck in a corner, and exit

Engineering Skills

- Subsystem testing again—build that good habit!
- Using an indicator LED to visibly display the state of a system input
- Using an indicator LED to visibly display program flow

Chapter 5 Challenges

Questions

1. What kind of electrical connection is a whisker?
2. When a whisker is pressed, what voltage occurs at the I/O pin monitoring it? What binary value will the **`digitalRead`** function return? If digital pin 8 is used to monitor the whisker circuit, what value does **`digitalRead`** return when a whisker is pressed, and what value does it return when a whisker is not pressed?
3. If **`digitalRead(7)== 1`**, what does that mean? What does it mean if **`digitalRead(7)== 0`**? How about **`digitalRead(5)== 1`** and **`digitalRead(5)== 0`**?

4. What statements did this chapter use to call different navigation functions based on whisker states?
5. What is the purpose of having nested **`if`** statements?

Exercises

1. Write a routine that uses a single variable named **`whiskers`** to track whisker contacts. It should store a 3 when no whiskers are contacted, a 2 if the right whisker is contacted, a 1 if the left whisker is contacted, or 0 if both whiskers are contacted. Hint: multiply the result by two.
2. Modify the **`loop`** function in RoamingWithWhiskers so that it makes the BOE Shield-Bot stop and not restart when both whiskers contact at the same time.
3. Add a function named **`pause`** to RoamingWithWhiskers. It should make the BOE Shield-Bot stay still for a certain amount of time.
4. Modify the **`loop`** function so that the BOE Shield-Bot stays still for 0.5 seconds before backing up and turning.

Projects

1. Modify RoamingWithWhiskers so that the BOE Shield-Bot stops and makes a 4 kHz beep that lasts 100 ms before executing its usual evasive maneuver. Make it beep twice if both whisker contacts are detected during the same sample. HINT: Use the **`pause`** function you developed in the Exercises section to make it pause immediately after the tone starts playing. Also, a 0.2 second pause after the tone call separates the 0.1 second tone from servo motion, or allows you to hear a second tone.
2. Modify RoamingWithWhiskers so that the BOE Shield-Bot roams in a 1 yard (or 1 meter) diameter circle. When you touch one whisker, it will cause the BOE Shield-Bot to travel in a tighter circle (smaller diameter). When you touch the other whisker, it will cause the BOE Shield-Bot to navigate in a wider diameter circle.

Question Solutions

1. A normally open, momentary, single-pole, single-throw tactile switch.
2. Zero (0) volts, resulting in binary zero (0) returned by **`digitalRead`**.
   `digitalRead(8)== 0` when whisker is pressed.
   `digitalRead(8)== 1` when whisker is not pressed.
3. **`digitalRead(7)== 1`** means the right whisker is not pressed.
   `digitalRead(7)== 0` means the right whisker is pressed.
   `digitalRead(5)== 1` means the left whisker is not pressed.
   `digitalRead(5)== 0` means the left whisker is pressed.
4. This chapter used **`if`**, **`if…else`**, and **`if…else if…else`** statements to evaluate whisker conditions and call navigation functions.

5. If one condition turns out to be true, the code might need to evaluate another condition with a nested **`if`** statement.

Exercise Solutions

1. Since **`digitalRead`** returns 1 or 0, your code can multiply **`digitalRead(5)`** by 2 and store the result in the **`whiskers`** variable. It can then add the result of **`digitalRead(7)`** to the **`whiskers`** variable and the result will be 3 for no whiskers.

```
// Robotics with the BOE Shield Chapter 5, Exercise 1
// Value from 0 to 3 indicates whisker states:
// 0 = both, 1 = left, 2 = right, 3 = neither.

void setup()                      // Built-in initialization block
{
  tone(4, 3000, 1000);            // Play tone for 1 second
  delay(1000);                    // Delay to finish tone

  pinMode(7, INPUT);              // Set right whisker pin to input
  pinMode(5, INPUT);              // Set left whisker pin to input

  Serial.begin(9600);             // Set data rate to 9600 bps
}

void loop()                       // Main loop auto-repeats
{
  byte whiskers = 2 * digitalRead(5);
  whiskers += digitalRead(7);

  Serial.println(whiskers);       // Display wLeft

  delay(50);                      // Pause for 50 ms
}
```

2. In the **`if((wLeft == 0)  && (wRight == 0))`** block, remove the **`backward`** and **`turnLeft`** function and replace them with calls to **`servoLeft.detach`** and **`servoRight.detach`**.

```
void loop()                           // Main loop auto-repeats
{
  byte wLeft = digitalRead(5);        // Copy right result to wLeft
  byte wRight = digitalRead(7);       // Copy left result to wRight

  if((wLeft == 0) && (wRight == 0))   // If both whiskers contact
  {
    pause(500);                       // Pause motion for 0.5 seconds
```

```
    backward(1000);                     // Back up 1 second
    turnLeft(800);                      // Turn left about 120 degrees
  }
  else if(wLeft == 0)                   // If only left whisker contact
  {
    pause(500);                         // Pause motion for 0.5 seconds
    backward(1000);                     // Back up 1 second
    turnRight(400);                     // Turn right about 60 degrees
  }
  else if(wRight == 0)                  // If only right whisker contact
  {
    pause(500);                         // Pause motion for 0.5 seconds
    backward(1000);                     // Back up 1 second
    turnLeft(400);                      // Turn left about 60 degrees
  }
  else                                  // Otherwise, no whisker contact
  {
    forward(20);                        // Forward 1/50 of a second
  }
}
```

3. Solution:

```
void pause(int time)                    // Pause drive wheels
{
  servoLeft.writeMicroseconds(1500);    // Left wheel stay still
  servoRight.writeMicroseconds(1500);   // Right wheel stay still
  delay(time);                          // Maneuver for time ms
}
```

4. Make sure not to call this **pause** function in the **else** condition because the **forward** function is only supposed to go forward for 20 ms before checking the whiskers again.

```
void loop()                             // Main loop auto-repeats
{
  byte wLeft = digitalRead(5);          // Copy right result to wLeft
  byte wRight = digitalRead(7);         // Copy left result to wRight

  if((wLeft == 0) && (wRight == 0))     // If both whiskers contact
  {
    pause(500);                         // Pause motion for 0.5 seconds
    backward(1000);                     // Back up 1 second
    turnLeft(800);                      // Turn left about 120 degrees
  }
  else if(wLeft == 0)                   // If only left whisker contact
  {
    pause(500);                         // Pause motion for 0.5 seconds
    backward(1000);                     // Back up 1 second
```

```
    turnRight(400);                     // Turn right about 60 degrees
  }
  else if(wRight == 0)                  // If only right whisker contact
  {
    pause(500);                         // Pause motion for 0.5 seconds
    backward(1000);                     // Back up 1 second
    turnLeft(400);                      // Turn left about 60 degrees
  }
  else                                  // Otherwise, no whisker contact
  {
    forward(20);                        // Forward 1/50 of a second
  }
}
```

Project Solutions

1. The key to solving this problem is to write a statement that makes a beep with the required parameters. As soon as the beep starts, call the **pause** function to keep the BOE Shield-Bot still while it beeps. Make sure not to add any **pause** calls to the **else** statement's code block. It needs to repeatedly go forward for 20 ms, without any pauses.

```
// RoamingWithWhiskers  Chapter 5 Project 1
// Go forward.  Back up and turn if whiskers indicate BOE Shield bot
// bumped into something.

#include <Servo.h>                  // Include servo library

Servo servoLeft;                    // Declare left and right servos
Servo servoRight;

void setup()                        // Built-in initialization block
{
  pinMode(7, INPUT);                // Set right whisker pin to input
  pinMode(5, INPUT);                // Set left whisker pin to input

  tone(4, 3000, 1000);              // Play tone for 1 second
  delay(1000);                      // Delay to finish tone

  servoLeft.attach(13);             // Attach left signal to pin 13
  servoRight.attach(12);            // Attach right signal to pin 12
}

void loop()                         // Main loop auto-repeats
{
  byte wLeft = digitalRead(5);      // Copy right result to wLeft
  byte wRight = digitalRead(7);     // Copy left result to wRight

  if((wLeft == 0) && (wRight == 0))// If both whiskers contact
  {
```

```
    tone(4, 4000, 100);           // Play a 0.1 ms tone
    pause(200);                   // Stop for 0.2 seconds
    tone(4, 4000, 100);           // Play a 0.1 ms tone
    pause(200);                   // Stop for 0.2 seconds
    backward(1000);               // Back up 1 second
    turnLeft(800);                // Turn left about 120 degrees
  }
  else if(wLeft == 0)             // If only left whisker contact
  {
    tone(4, 4000, 100);           // Play a 0.1 ms tone
    pause(200);                   // Stop for 0.2 seconds
    backward(1000);               // Back up 1 second
    turnRight(400);               // Turn right about 60 degrees
  }
  else if(wRight == 0)            // If only right whisker contact
  {
    tone(4, 4000, 100);           // Play a 0.1 ms tone
    pause(200);                   // Stop for 0.2 seconds
    backward(1000);               // Back up 1 second
    turnLeft(400);                // Turn left about 60 degrees
  }
  else                            // Otherwise, no whisker contact
  {
    forward(20);                  // Forward 1/50 of a second
  }
}

void pause(int time)                  // Backward function
{
  servoLeft.writeMicroseconds(1500);  // Left wheel clockwise
  servoRight.writeMicroseconds(1500); // Right wheel counterclockwise
  delay(time);                        // Maneuver for time ms
}

void forward(int time)                // Forward function
{
  servoLeft.writeMicroseconds(1700);  // Left wheel counterclockwise
  servoRight.writeMicroseconds(1300); // Right wheel clockwise
  delay(time);                        // Maneuver for time ms
}

void turnLeft(int time)               // Left turn function
{
  servoLeft.writeMicroseconds(1300);  // Left wheel clockwise
  servoRight.writeMicroseconds(1300); // Right wheel clockwise
  delay(time);                        // Maneuver for time ms
}

void turnRight(int time)              // Right turn function
{
  servoLeft.writeMicroseconds(1700);  // Left wheel counterclockwise
  servoRight.writeMicroseconds(1700); // Right wheel counterclockwise
  delay(time);                        // Maneuver for time ms
}
```

```
void backward(int time)                // Backward function
{
  servoLeft.writeMicroseconds(1300);   // Left wheel clockwise
  servoRight.writeMicroseconds(1700);  // Right wheel counterclockwise
  delay(time);                         // Maneuver for time ms
}
```

2. Start with the Circle sketch, from the Chapter 4 Project Solutions that begin on page 144. Comment the **detach** calls and move the circle code to the **loop** function and reduce the **delay** to 50 ms so that it can check the whiskers for contacts 20 times per second. Then, add the whisker monitoring code with an **if** statement that reduces or increases a variable that slows the right wheel when the right whisker is pressed, or speeds up the right wheel if the left whisker is pressed.

```
// Robotics with the BOE Shield - Chapter 5, project 2 – WhiskerCircle
// BOE Shield-Bot navigates a circle of 1 yard diameter.
// Tightens turn if right whisker pressed; reduces turn if left whisker
// is pressed.

#include <Servo.h>                     // Include servo library

Servo servoLeft;                       // Declare left and right servos
Servo servoRight;

int turn;

void setup()                           // Built-in initialization block
{
  pinMode(7, INPUT);                   // Set right whisker pin to input
  pinMode(5, INPUT);                   // Set left whisker pin to input

  tone(4, 3000, 1000);                 // Play tone for 1 second
  delay(1000);                         // Delay to finish tone

  servoLeft.attach(13);                // Attach left signal to Port 13
  servoRight.attach(12);               // Attach right signal to Port 12

  turn = 0;

//  servoLeft.detach();                // Stop sending servo signals
//  servoRight.detach();
}
```

```
void loop()                            // Main loop auto-repeats
{                                      // Nothing needs repeating
  int wLeft = digitalRead(5);
  int wRight = digitalRead(7);

  if(wLeft == 0)
  {
    turn -= 10;
  }
  else if(wRight == 0)
  {
    turn += 10;
  }

  // Arc to the right
  servoLeft.writeMicroseconds(1600);  // Left wheel counterclockwise
  servoRight.writeMicroseconds(1438 + turn); // Rt wheel clockwise slower
  delay(50);                           // ...for 25.5 seconds
}
```

Chapter 6. Light-Sensitive Navigation with Phototransistors

Light sensors have many applications in robotics and industrial control: finding the edge of a roll of fabric in a textile mill, determining when to activate streetlights at different times of the year, when to take a picture, or when to deliver water to a crop of plants.

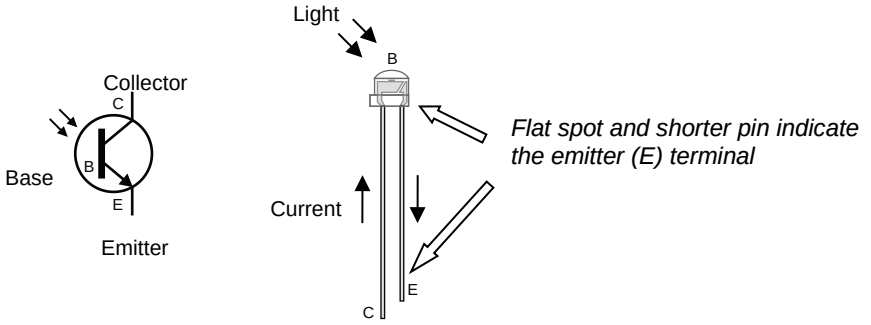
The light sensors in your Robotics Shield Kit respond to visible light, and also to an invisible type of light called *infrared*. These sensors can be used in different circuits that the Arduino can monitor to detect variations in light level. With this information, your sketch can be expanded to make the BOE Shield-Bot navigate by light, such as driving toward a flashlight beam or an open doorway letting light into a dark room.

Light-sensitive phototransistors let the BOE Shield-Bot navigate by driving towards a bright light source.

Introducing the Phototransistor

A *transistor* is like a valve that regulates the amount of electric current that passes through two of its three terminals. The third terminal controls just how much current passes through the other two. Depending on the type of transistor, the current flow can be controlled by voltage, current, or in the case of the *phototransistor*, by light.

The drawing below shows the schematic and part drawing of the phototransistor in your Robotics Shield Kit. The brightness of the light shining on the phototransistor's base (B) terminal determines how much current it will allow to pass into its collector (C) terminal, and out through its emitter (E) terminal. Brighter light results in more current; less-bright light results in less current.

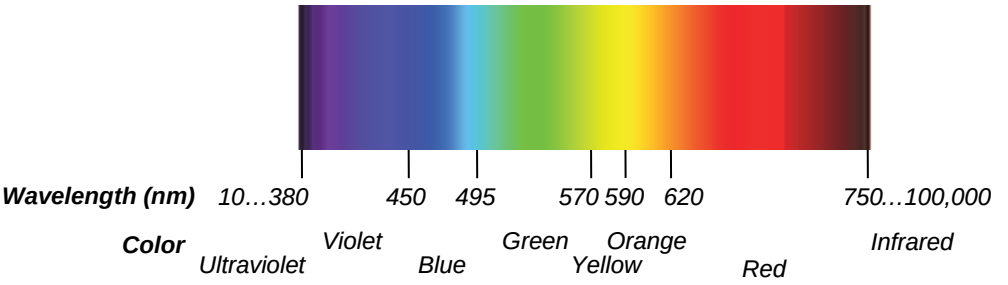


The phototransistor looks a little bit like an LED. The two devices do have two similarities. First, if you connect the phototransistor in the circuit backwards, it won't work right. Second, it also has two different length pins and a flat spot on its plastic case for identifying its terminals. The longer of the two pins indicates the phototransistor's collector terminal. The shorter pin indicates the emitter, and it connects closer to a flat spot on the phototransistor's clear plastic case.

Light Waves

In the ocean, you can measure the distance between the peaks of two adjacent waves in feet or meters. With light, which also travels in waves, the distance between adjacent peaks is measured in *nanometers* (nm) which are billionths of meters. The figure below shows the wavelengths for colors of light we are familiar with, along with some the human eye cannot detect, such as ultraviolet and infrared.

You can see this image in color in the free PDF version of this book, availabe for download from the #122-32335 product page at www.parallax.com.



The phototransistor in the Robotics Shield Kit is most sensitive to 850 nm wavelengths, which is in the infrared range. Infrared light is not visible to the human eye, but many different light sources emit considerable amounts of it, including halogen and incandescent lamps, and especially the sun. This phototransistor also responds to visible light, though it's less sensitive, especially to wavelengths below 450 nm.

The phototransistor circuits in this chapter are designed to work well indoors, with fluorescent or incandescent lighting. Make sure to avoid direct sunlight and direct halogen lights; they would flood the phototransistors with too much infrared light.

In your robotics area, close window blinds to block direct sunlight, and point any halogen lamps upward so that the light is reflected off the ceiling.

Activity 1: Simple Light to Voltage Sensor

Imagine that your BOE Shield-Bot is navigating a course, and there's a bright light at the end. Your robot's final task in the course is to stop underneath that bright light. There's a simple phototransistor circuit you can use that lets the Arduino know it detected bright light with a binary-1, or ambient light with a binary-0. Incandescent bulbs in desk lamps and flashlights make the best bright-light sources. Compact fluorescent and LED light sources are not as easy for the circuit in this activity to recognize.

Ambient means 'existing or present on all sides' according to Merriam Webster's dictionary. For the light level in a room, think about ambient light as the overall level of brightness.

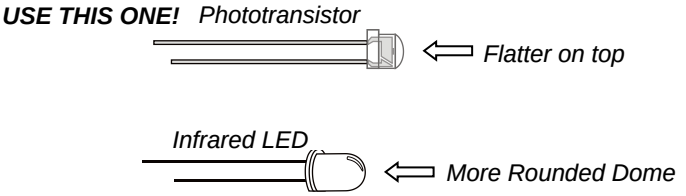
Parts List

(1) phototransistor
(2) jumper wires
(1) resistor, 2 kΩ (red-black-red)
(1) incandescent or fluorescent flashlight or desk lamp

After some testing, and depending on the light conditions in your robotics area, you might end up replacing the 2 kΩ resistor with one of these resistors, so keep them handy:

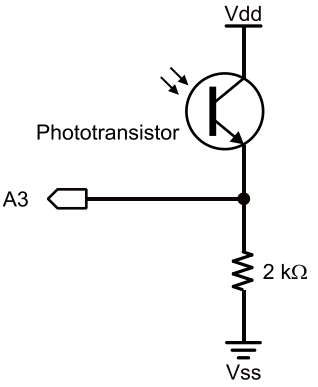
(1) resistor, 220 Ω (red-red-brown)
(1) resistor, 470 Ω (yellow-violet-brown)
(1) resistor, 1 kΩ (brown-black-red)
(1) resistor, 4.7 kΩ (yellow-violet-red)
(1) resistor, 10 kΩ (brown-black-orange)

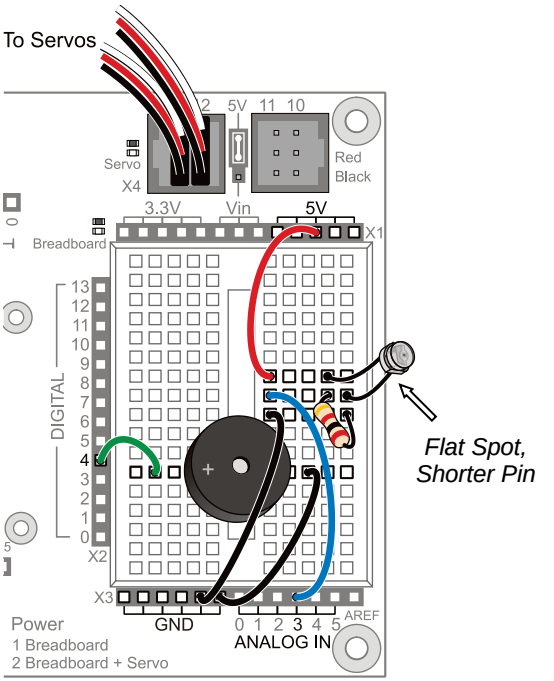
The next drawing will help you tell apart the phototransistor and infrared LED, since they look similar.



Building the Bright Light Detector

- ✓ The schematic and wiring diagram below show a circuit very similar to the ones in streetlights that turn on automatically at night. The circuit outputs a voltage that varies depending on how much light shines on the phototransistor. The Arduino will monitor the voltage level with one of its analog input pins.

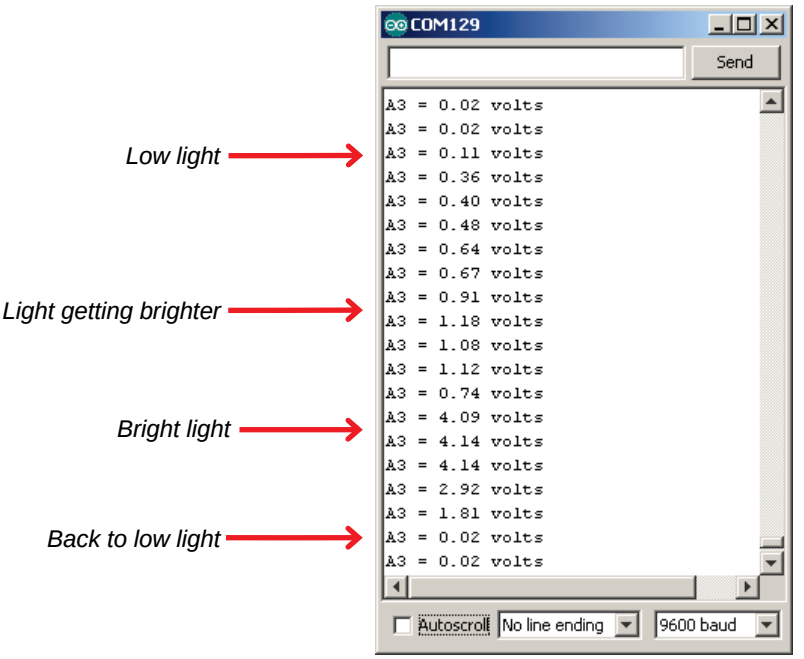




- ✓ Disconnect the battery pack and programming cable from your Arduino, and set the BOE Shield's switch to 0.
- ✓ Remove the whisker circuits, but leave the piezospeaker circuit in place.
- ✓ Build the circuit shown, using the 2 kΩ resistor.
- ✓ Double-check to make sure you connect the phototransistor's emitter lead (by the flat spot) to the resistor, and its collector to 5V.
- ✓ Also double-check that the phototransistor's leads are not touching each other.

Example Sketch: PhototransistorVoltage

The PhototransistorVoltage sketch makes the Serial Monitor display the voltage measured at A3—one of the Arduino's five analog input channels that are accessible through the BOE Shield. In the circuit you just built, a wire connects A3 to the row where the phototransistor's emitter and resistor meet. The voltage at this part of the circuit will change as the light level sensed by the phototransistor changes. The Serial Monitor screen capture below shows some example voltage measurements.



- ✓ Reconnect programming cable and battery pack power to your board.
- ✓ Put the BOE Shield's power switch in position 1.
- ✓ Create and save the PhototransistorVoltage sketch, and run it on your Arduino.
- ✓ Slowly move the flashlight or lamp over the phototransistor, and watch the value of A3 in the Serial Monitor. Brighter light should cause larger voltage values and dimmer light should cause smaller voltages.
- ✓ If the ambient light is brighter than just fluorescent lights, and you have a bright flashlight, you may need to replace the 2 kΩ resistor with a smaller value. Try 1 kΩ, 470 Ω, or even 220 Ω for really bright lights.
- ✓ If the ambient light is low, and you are using a fluorescent desk lamp bulb or an LED flashlight for your bright light, you may need to change the 2 kΩ resistor to 4.7 kΩ, or even 10 kΩ.
- ✓ Record values for ambient light (your normal room light levels), and then bright light, like when you shine a flashlight right on the phototransistor. Make a note of them—you'll need them for the sketch after this one.

```
/*
 * Robotics with the BOE Shield - PhototransistorVoltage
 * Display voltage of phototransistor circuit output connected to A3 in
 * the serial monitor.
 */
```

```
void setup()                              // Built-in initialization block
{
  Serial.begin(9600);                     // Set data rate to 9600 bps
}

void loop()                               // Main loop auto-repeats
{
  Serial.print("A3 = ");                  // Display "A3 = "
  Serial.print(volts(A3));                // Display measured A3 volts
  Serial.println(" volts");               // Display " volts" & newline
  delay(1000);                            // Delay for 1 second
}

float volts(int adPin)                    // Measures volts at adPin
{                                         // Returns floating point voltage
 return float(analogRead(adPin)) * 5.0 / 1024.0;
}
```

Halt Under the Bright Light

The sketch HaltUnderBrightLight will make the BOE Shield-Bot go forward until the phototransistor detects light that's bright enough to make the voltage applied to A3 exceed 3.5 V. You can change the 3.5 V value to one that's halfway between the high and low voltage values you recorded from the last sketch.

- ✓ Calculate the half-way point between the ambient and bright light voltages you recorded from the last sketch.
- ✓ In the HaltUnderBrightLight sketch, substitute your half way point value in place of 3.5 in the statement **`if(volts(A3) > 3.5`**.
- ✓ Run your modified version of HaltUnderBrightLight on the Arduino.
- ✓ Hold your flashlight or lamp about a foot off of the floor, and put the BOE Shield-Bot on the floor a couple feet away but pointed so it will go straight under the light.
- ✓ Move the power switch to position 2 so the BOE Shield-Bot will drive forward. How close did it get to stopping directly under the light?
- ✓ Try making adjustments to the threshold you set in the **`if(volts(A3) >…)`** statement to get the BOE Shield-Bot to park right underneath that bright light.

```
/*
 * Robotics with the BOE Shield - HaltUnderBrightLight
 * Display voltage of phototransistor circuit output connected to A3 in
 * the serial monitor.
 */
```

```
#include <Servo.h>                        // Include servo library

Servo servoLeft;                          // Declare left and right servos
Servo servoRight;

void setup()                              // Built-in initialization block
{
  tone(4, 3000, 1000);                    // Play tone for 1 second
  delay(1000);                            // Delay to finish tone

  servoLeft.attach(13);                   // Attach left signal to pin 13
  servoRight.attach(12);                  // Attach right signal to pin 12

  servoLeft.writeMicroseconds(1700);      // Full speed forward
  servoRight.writeMicroseconds(1300);
}

void loop()                               // Main loop auto-repeats
{
  if(volts(A3) > 3.5)                     // If A3 voltage greater than 3.5
  {
    servoLeft.detach();                   // Stop servo signals
    servoRight.detach();
  }
}

float volts(int adPin)                    // Measures volts at adPin
{                                         // Returns floating point voltage
 return float(analogRead(adPin)) * 5.0 / 1024.0;
}
```

How the Volts Function Works

The Arduino's A0, A1…A5 sockets are connected to Atmel microcontroller pins that are configured for *analog to digital conversion*. It's how microcontrollers measure voltage: they split a voltage range into many numbers, with each number representing a voltage. Each of the Arduino's analog inputs has a 10-bit *resolution*, meaning that it uses 10 binary digits to describe its voltage measurement. With 10 binary digits, you can count from 0 to 1023; that's a total of 1024 voltage levels if you include zero.

By default, the Arduino's **`analogRead`** function is configured to use the 0…1023 values to describe where a voltage measurement falls in a 5 V scale. If you split 5 V into 1024 different levels, each level is 5/1024^{ths} of a volt apart. 5/1024^{ths} of a volt is approximately 0.004882813 V, or about 4.89 thousandths of a volt. So, to convert a value returned by **`analogRead`** to a voltmeter-style value, all the volts function has to do is multiply by 5 and divide by 1024.

Example: the **analogRead** function returns 645; how many volts is that? Answer:

$$V = 645 \times \frac{5V}{1024}$$
$$= 3.1494140625\ V$$
$$\approx 3.15\ V$$

The sketches have been calling the **volts** function with **volts(A3)**. When they do that, they pass **A3** to its ***adPin*** parameter. Inside the function, **analogRead(*adPin*)** becomes **analogRead(A3)**. It returns a value in the 0 to 1023 range that represents the voltage applied to A3. The **analogRead** call returns an integer, but since it is nested in **float(analogRead(*adPin*))**, that integer value gets converted to floating point. Then, it's multiplied by the floating point value 5.0 and divided by 1024.0, which converts it to a voltmeter value (just like we converted 645 to 3.15 V).

```
float volts(int adPin)                  // Measures volts at adPin
{                                       // Returns floating point voltage
 return float(analogRead(adPin)) * 5.0 / 1024.0;
}
```

Since **return** is to the left of the calculation in the **volts** function block, the result gets returned to the function call. The sketch PhototransistorVoltage displays the value returned by the **volts** function with **Serial.print(volts(A3))**.

HaltUnderBrightLight uses that value in the **if(volts(A3) > 3.5)** expression to bring the BOE Shield-Bot to a halt under the light.

Binary vs. Analog and Digital

A binary sensor can transmit two different states, typically to indicate the presence or absence of something. For example, a whisker sends a high signal if it is not pressed, or a low signal if it is pressed.

An analog sensor sends a continuous range of values that correspond to a continuous range of measurements. The phototransistor circuits in this activity are examples of analog sensors. They provide continuous ranges of values that correspond to continuous ranges of light levels.

A *digital value* is a number expressed by digits. Computers and microcontrollers store analog measurements as digital values. The process of measuring an analog sensor and storing that measurement as a digital value is called analog to digital conversion. The measurement is called a digitized measurement. Analog to digital conversion documents will also call them quantized measurements.

The Arduino Map Function

In the PhototransistorVoltage sketch, we converted measurements from the 0 to 1023 range to the 0.0 to 4.995 volt range for display. For other applications, you might need to convert the value to some other range of integers so that your sketch can pass it to another function, maybe for motor control, or maybe for more analysis.

That's where the Arduino's **map** function comes in handy. It's useful for "mapping" a value in one range of integers to an equivalent value in some other range. For example, let's say you want to map a measurement in the 0 to 1024 range to a range of 1300 to 1700 for servo control. Here is an example of how you could use the **map** function to do it:

```
int adcVal = analogRead(A3);
int newAdcVal = map(adcVal, 0, 1023, 1300, 1700);
```

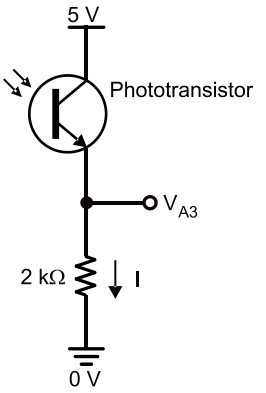
In this example, if the value of **adcVal** is 512 , the result of the **map** function for the **newAdcVal** call would be 1500 . So, the measurement got mapped from a point about half way through the 0..1023 range to its equivalent point in the 1300...1700 range.

How the Phototransistor Circuit Works

A resistor "resists" the flow of current. Voltage in a circuit with a resistor can be likened to water pressure. For a given amount of electric current, more voltage (pressure) is lost across a larger resistor than a smaller resistor that has the same amount of current passing through it. If you instead keep the resistance constant and vary the current, you can measure a larger voltage (pressure drop) across the same resistor with more current, or less voltage with less current.

The Arduino's analog inputs are invisible to the phototransistor circuit. So, A3 monitors the circuit but has no effect on it. Take a look at the circuit below. With 5 volts (5 V) at the top and GND (0 V) at the bottom of the circuit, 5 V of electrical pressure (voltage) makes the supply of electrons in the BOE Shield-Bot's batteries want to flow through it.

The reason the voltage at A3 (V_{A3}) changes with light is because the phototransistor lets more current pass when more light shines on it, or less current pass with less light. That current, which is labeled I in the circuit below, also has to pass through the resistor. When more current passes through a resistor, the voltage across it will be higher. When less current passes, the voltage will be lower. Since one end of the resistor is tied to GND = 0 V, the voltage at the V_{A3} end goes up with more current and down with less current.



If you replace the 2 kΩ resistor with a 1 kΩ resistor, V_{A3} will see smaller values for the same currents. In fact, it will take twice as much current to get V_{A3} to the same voltage level, which means the light will have to be twice as bright to reach the 3.5 V level, the default voltage in HaltUnderBrightLight to make the BOE Shield-Bot stop.

So, a smaller resistor in series with the phototransistor makes the circuit less sensitive to light. If you instead replace the 2 kΩ resistor with a 10 kΩ resistor, V_{A3} will be 5 times larger with the same current, and it'll only take $1/5^{th}$ the light to generate $1/5^{th}$ the current to get V_{A3} past the 3.5 V level. So, a larger resistor makes the circuit more sensitive to light.

Connected in Series: When two or more elements are connected end-to-end, they are *connected in series*. The phototransistor and resistor in this circuit are connected in series.

Ohm's Law

Two properties affect the voltage at V_{A3}: current and resistance, and Ohm's Law explains how it works. Ohm's Law states that the voltage (V) across a resistor is equal to the current (I) passing through it multiplied by its resistance (R). So, if you know two of these values, you can use the Ohm's Law equation to calculate the third:

$$\mathbf{V = I \times R}$$

E = I x R: In some textbooks, you will see E = I × R instead. E stands for electric potential, which is another way to say "volts."

Voltage (V) is measured in units of volts, which are abbreviated with an upper-case V. Current (I) is measured in amperes, or amps, which are abbreviated A. Resistance (R) is

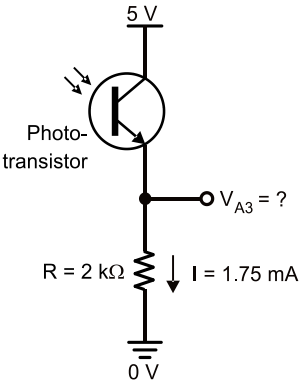
measured in ohms which is abbreviated with the Greek letter omega (Ω). The current levels you are likely to see through this circuit are in milliamps (mA). The lower-case m indicates that it's a measurement of thousandths of amps. Similarly, the lower-case k in kΩ indicates that the measurement is in thousands of ohms.

Let's use Ohm's Law to calculate V_{A3} in with the phototransistor, letting two different amounts of current flow through the circuit:

- 1.75 mA, which might happen as a result of fairly bright light
- 0.25 mA, which would happen with less bright light

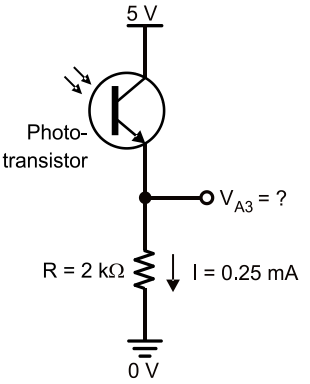
The examples below show the conditions and their solutions. When you try these calculations, remember that milli (m) is thousand*ths* and kilo (k) is thousand*s* when you substitute the numbers into Ohm's Law.

Example 1: I = 1.75 mA and R = 2 kΩ



$$
\begin{aligned}
V_{A3} &= I \times R \\
&= 1.75 mA \times 2 k\Omega \\
&= \frac{1.75}{1000} A \times 2000 \Omega \\
&= 1.75 A \times 2 \Omega \\
&= 3.5 A\Omega \\
&= 3.5 V
\end{aligned}
$$

Example 2: 1 = 0.25 mA and R = 2 kΩ



$$
\begin{aligned}
V_{A3} &= I \times R \\
&= 0.25mA \times 2k\Omega \\
&= \frac{0.25}{1000} A \times 2000\,\Omega \\
&= 0.25A \times 2\Omega \\
&= 0.5\,A\Omega \\
&= 0.5\,V
\end{aligned}
$$

Your Turn – Ohm's Law and Resistor Adjustments

Let's say that the ambient light in your room is twice as bright as the light that resulted in V_{A3} = 3.5 V for bright light and 0.5 V for shade. Another situation that could cause higher current is if the ambient light is a stronger source of infrared. In either case, the phototransistor could allow twice as much current to flow through the circuit, which could lead to measurement difficulties.

- **Question**: What could you do to bring the circuit's voltage response back down to 3.5 V for bright light and 0.5 V for dim?
- **Answer**: Cut the resistor value in half; make it 1 kΩ instead of 2 kΩ.

✓ Try repeating the Ohm's Law calculations with R = 1 kΩ, bright current I = 3.5 mA, and dim current I = 0.5 mA. Does it bring V_{A3} back to 3.5 V for bright light and 0.5 V for dim light with twice the current? (It should; if it didn't for you, check your calculations.)

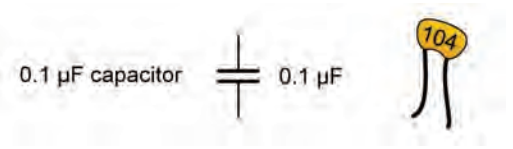
Activity 2: Measure Light Levels Over a Larger Range

The circuit in the previous activity only works over a limited light range. You might get the Activity #1 circuit all nice and calibrated in one room, then take it to a brighter room and find that all the voltage measurements will sit at the maximum value. Or, maybe you'll take it into a darker room, and the voltages will end up never making it past 0.1 V.

This activity introduces a different phototransistor circuit that the Arduino can use to measure a much wider range of light levels. This circuit and sketch can return values ranging from 0 to over 75,000. **Be aware: this time the smaller values indicate bright light, and large values indicate low light.**

Introducing the Capacitor

A *capacitor* is a device that stores charge, and it is a fundamental building block of many circuits. Batteries are also devices that store charge, and for these activities it will be convenient to think of capacitors as tiny batteries that can be charged, discharged, and recharged.



How much charge a capacitor can store is measured in farads (F). A *farad* is a very large value that's not practical for use with these BOE Shield-Bot circuits. The capacitors in your kit store fractions of millionths of farads. A millionth of a farad is called a *microfarad*, and it is abbreviated µF. This one stores one tenth of one millionth of a farad: 0.1 µF.

Common Capacitance Measurements

microfarads:	(millionths of a farad), abbreviated µF	1 µF = 1×10^{-6} F
nanofarads:	(billionths of a farad), abbreviated nF	1 nF = 1×10^{-9} F
picofarads:	(trillionths of a farad), abbreviated pF	1 pF = 1×10^{-12} F

The 104 on the 0.1 µF capacitor's case is a measurement in picofarads (pF). In this labeling system, 104 is the number 10 with four zeros added, so the capacitor is 100,000 pF, or 0.1 µF.

$(100{,}000) \times (1 \times 10^{-12})$ F = $(100 \times 10^{3}) \times (1 \times 10^{-12})$ F
= 100×10^{-9} F = 0.1×10^{-6} F
= 0.1 µF.

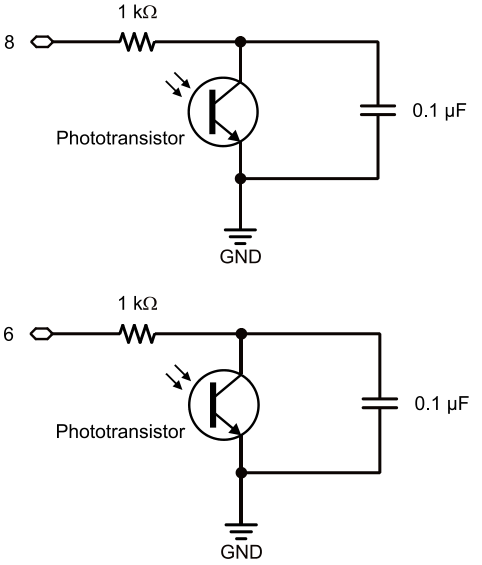
Building the Photosensitive Eyes

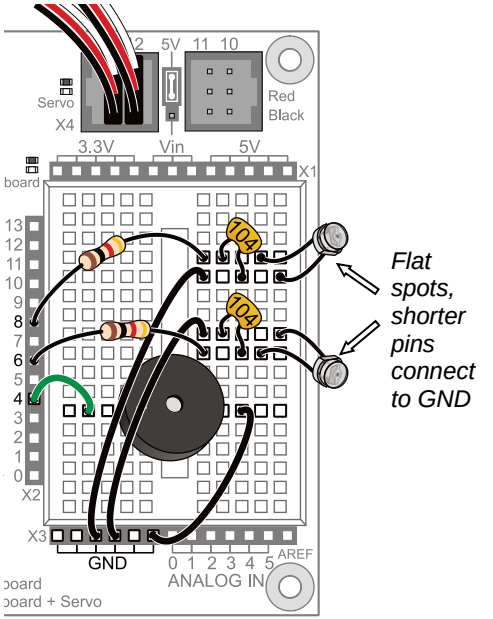
These circuits can respond independently to the light level reaching each phototransistor. They will be pointing upward at about 45°, one forward-left and the other forward-right. This way, a sketch monitoring the values of both phototransistors can determine which side of the BOE Shield-Bot sees brighter light. Then, this information can be used for navigation decisions.

Parts List

(2) phototransistors
(2) capacitors, 0.1 µF (104)
(2) resistors, 1 kΩ (brown-black-red)
(2) jumper wires

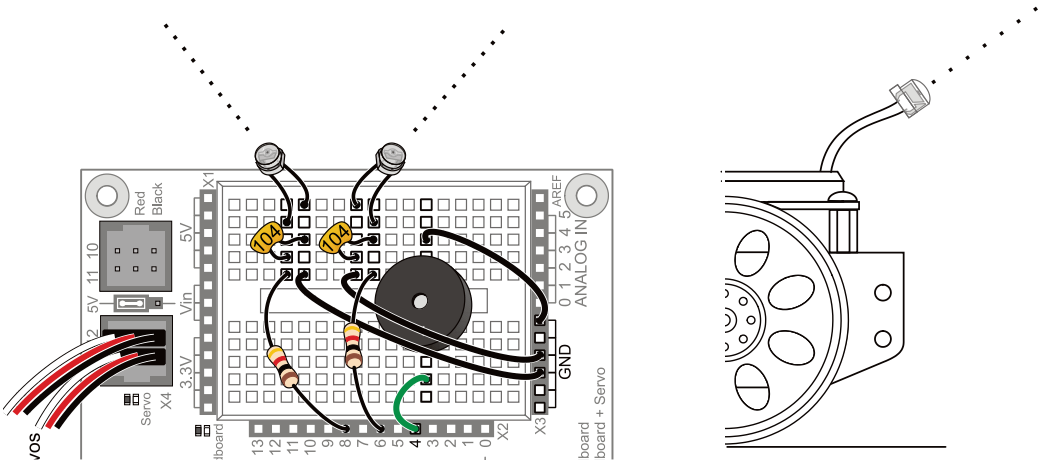
- ✓ Disconnect batteries and programming cable from your board.
- ✓ Remove the old phototransistor circuit, and build the circuits shown below.
- ✓ Double-check your circuits against the wiring diagram to make sure your phototransistors are not plugged in backwards, and that the leads are not touching.





The roaming examples in this chapter will depend on the phototransistors being pointed upward and outward to detect differences in light levels from different directions.

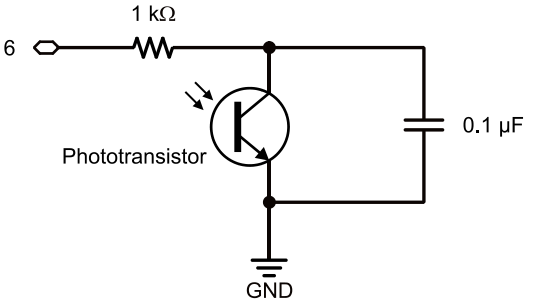
- ✓ Adjust the phototransistors to point upward at a 45° from the breadboard, and outward about 90° apart, as shown next.



About Charge Transfer and the Phototransistor Circuit

Think of each capacitor in this circuit as a tiny rechargeable battery, and think of each phototransistor as a light-controlled current valve. Each capacitor can be charged to 5 V and then allowed to drain through its phototransistor. The rate that the capacitor loses its charge depends on how much current the phototransistor (current valve) allows to pass, which in turn depends on the brightness of the light shining on the phototransistor's base. Again, brighter light results in more current passing. Shadows result in less current.

This kind of phototransistor/capacitor circuit is called a *charge transfer* circuit. The Arduino will determine the rate at which each capacitor loses its charge through its phototransistor by measuring how long it takes the capacitor's voltage to *decay*, that is, to drop below a certain voltage value. The decay time corresponds to how wide open that current valve is, which is controlled by the brightness of the light reaching the phototransistor's base. More light means faster decay, less light means slower decay.

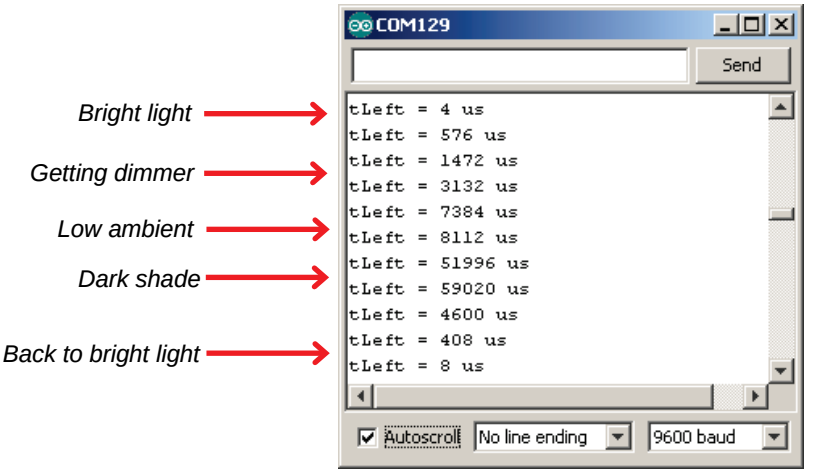


QT Circuit: A common abbreviation for charge transfer is QT. The letter Q refers to electrical charge (an accumulation of electrons), and T is for transfer.

Connected in Parallel: The phototransistor and capacitor shown above are connected in parallel; each of their leads are connected to common terminals (also called nodes). The phototransistor and the capacitor each have one lead connected to GND, and they also each have one lead connected to the same 1 kΩ resistor lead.

Test the Phototransistor Circuit

The sketch LeftLightSensor charges the capacitor in the pin 8 QT circuit, measures the voltage decay time, and displays it in the Serial Monitor. Remember, with this circuit and sketch, lower numbers mean brighter light.



We'll be using this light-sensing technique for the rest of the chapter, so you can take the BOE Shield-Bot from one room to another without having to worry about finding the right resistors for different ambient light levels.

- ✓ If there is direct sunlight shining in through the windows, close the blinds.
- ✓ Create, save, and run LeftLightSensor, and open the Serial Monitor.
- ✓ Make a note of the value displayed in the Serial Monitor.
- ✓ If the Serial Monitor does not display values or seems to get stuck after just one or two, it may mean that there's an error in your circuit. If you see these symptoms, check your wiring and try again.
- ✓ Use your hand or a book to cast a shadow over the pin 8 phototransistor circuit.
- ✓ Check the measurement in the Serial Monitor again. The value should be larger than the first one. Make a note of it too.

- ✓ If there's no output to the Serial Monitor, or if it is just stuck at one value regardless of light level, there could be a wiring error. Double-check your circuit (and your code too, if you hand-entered it.)
- ✓ Move the object casting the shadow closer to the top of the phototransistor to make the shadow darker. Make a note of the measurement.
- ✓ Experiment with progressively darker shadows, even cupping your hand over the phototransistor. (When it's really dark you may have to wait a few seconds for the measurement to finish.)

```
/*
 * Robotics with the BOE Shield - LeftLightSensor
 * Measures and displays microsecond decay time for left light sensor.
 */

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  Serial.begin(9600);                   // Set data rate to 9600 bps
}

void loop()                             // Main loop auto-repeats
{
  long tLeft = rcTime(8);               // Left rcTime -> tLeft

  Serial.print("tLeft = ");             // Display tLeft label
  Serial.print(tLeft);                  // Display tLeft value
  Serial.println(" us");                // Display tLeft units + newline

  delay(1000);                          // 1 second delay
}
                                        // rcTime function at pin
long rcTime(int pin)                    // ..returns decay time
{
  pinMode(pin, OUTPUT);                 // Charge capacitor
  digitalWrite(pin, HIGH);              // ..by setting pin ouput-high
  delay(1);                             // ..for 5 ms
  pinMode(pin, INPUT);                  // Set pin to input
  digitalWrite(pin, LOW);               // ..with no pullup
  long time  = micros();                // Mark the time
  while(digitalRead(pin));              // Wait for voltage < threshold
  time = micros() - time;               // Calculate decay time
  return time;                          // Return decay time
}
```

Your Turn: Test the Other Phototransistor Circuit

Before moving on to navigation, you'll need to run the same test on the right (pin 6) light sensor circuit. Both circuits have to be working well before you can move on to using them for navigation—there's that subsystem testing again!

- ✓ In the **rcTime** call, change the ***pin*** parameter from 8 to 6.
- ✓ Change all instances of **tLeft** to **tRight**.
- ✓ Run the sketch, and verify that the pin 6 light sensor circuit is working.
- ✓ It would also be nice to have a third sketch that tests both phototransistor circuits.
- ✓ Re-save the sketch as BothLightSensors, and update the comments.
- ✓ Replace the **loop** function with the one below.

Try rotating your BOE Shield-Bot until one side is pointing toward the brightest light source in the room and the other is pointing away from it. What is the largest difference you can get between **tLeft** and **tRight** in the Serial Monitor?

```
void loop()                             // Main loop auto-repeats
{
  long tLeft = rcTime(8);               // Left rcTime -> tLeft
  Serial.print("tLeft = ");             // Display tLeft label
  Serial.print(tLeft);                  // Display tLeft value
  Serial.print(" us      ");            // Display tLeft units

  long tRight = rcTime(6);              // Left rcTime -> tRight
  Serial.print("tRight = ");            // Display tRight label
  Serial.print(tRight);                 // Display tRight value
  Serial.println(" us");                // Display tRight units + newline

  delay(1000);                          // 1 second delay
}
```

rcTime and Voltage Decay

When light levels are low, the custom **rcTime** function might take time measurements too large for **int** or even **word** variables to store. The next step up in storage capacity is a **long** variable, which can store values from -2,147,483,648 to 2,147,483,647. So, the function definition **long rcTime(int *pin*)** is set up to make the function return a **long** value when it's done. It also needs to know which pin to measure.

```
long rcTime(int pin)
```

A charge transfer measurement takes seven steps:

(1) Set the I/O pin high to charge the capacitor.
(2) Wait long enough for the capacitor to charge.
(3) Change the I/O pin to input.
(4) Check the time.
(5) Wait for the voltage to decay and pass below the Arduino's 2.1 V threshold.
(6) Check the time again.
(7) Subtract the step-3 time from the step-6 time. That's the amount of time the decay took.

```
{
  pinMode(pin, OUTPUT);          // Step 1, part 1
  digitalWrite(pin, HIGH);       // Step 1, part 2
  delay(1);                      // Step 2
  pinMode(pin, INPUT);           // Step 3  part 1
  digitalWrite(pin, LOW);        // Step 3, part 2
  long time  = micros();         // Step 4
  while(digitalRead(pin));       // Step 5
  time = micros() - time;        // Step 6 & 7
  return time;
}
```

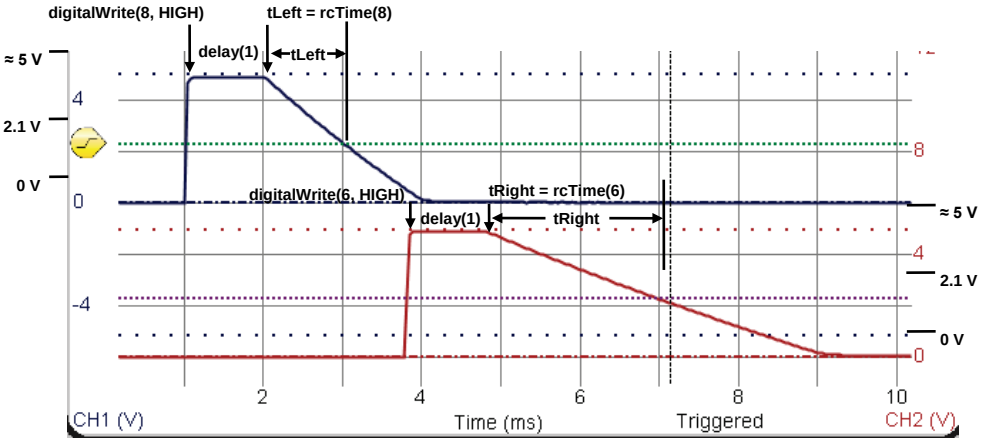
In this sketch, Step 1 has two sub-steps. First, **`pinMode(pin, OUPUT)`** sets the I/O pin to an output, then **`digitalWrite(pin, HIGH)`** makes it supply 5 V to the circuit.

Step 3 also has two sub-steps, because the I/O pin is sending a high signal. When the sketch changes the I/O pin's direction from output-high to input, it adds 10 kΩ of resistance to the circuit, which must be removed. Adding **`digitalWrite(pin, LOW)`** after **`pinMode(pin, INPUT)`** removes that resistance and allows the capacitor to drain its charge normally through the phototransistor.

Optional Advanced Topic: Voltage Decay Graphs

The next graph shows the BOE Shield-Bot's left and right QT circuit voltage responses while the BothLightSensors sketch is running. The device that measures and graphs these voltage responses over time is called an *oscilloscope*.

The two lines that graph the two voltage signals are called *traces*. The voltage scale for the upper trace is along the left, and the voltage scale for the lower trace is along the right. The time scale for both traces is along the bottom. Labels above each trace show when each command in BothLightSensors executes, so that you can see how the voltage signals respond.



The upper trace in the graph plots the capacitor's voltage in the pin 8 QT circuit; that's the left light sensor. In response to **`digitalWrite(8, HIGH)`**, the voltage quickly rises from 0 V to almost 5 V at about the 1 ms mark. The signal stays at around 5 V for the duration of **`delay(1)`**. Then, at the 2 ms mark, the **`rcTime`** call causes the decay to start. The **`rcTime function`** measures the time it takes the voltage to decay to about 2.1 V and stores it in the **`tLeft`** variable. In the plot, it looks like that decay took about 1 ms, so the **`tLeft`** variable should store a value close to 1000.

The lower trace in the graph plots the pin 6 QT circuit's capacitor voltage—the right light sensor. This measurement starts after the left sensor measurement is done. The voltage varies in a manner similar to the upper trace, except the decay time takes about 2 ms. We would expect to see **`tRight`** store a value in the 2000 neighborhood. This larger value corresponds to a slower decay, which in turn corresponds to a lower light level.

Activity 3: Light Measurements for Roaming

We now have circuits that can work under a variety of lighting conditions. Now we need some code that can adapt as well. An example of sketch code that cannot adapt to change would be:

```
if(tLeft > 2500)…          // Not good for navigation.
```

Maybe that statement would work well for turning away from shadows in one room, but take it to another with brighter lights, and it might never detect a shadow. Or, take it to a darker room, and it might think it's seeing shadows all the time. For navigation, what matters is not an actual number reporting the light level over each sensor. What matters is the *difference* in how much light the two sensors detect, so the robot can turn toward the sensor seeing brighter light (or away from it, depending on what you want.)

The solution is simple. Just divide the right sensor measurement into the sum of both. Your result will always be in the 0 to 1 range. This technique is an example of a *normalized differential* measurement Here's what it looks like as an equation:

$$normalized\ differential\ shade = \frac{tRight}{tRight + tLeft}$$

For example, a normalized differential measurement of 0.25 would mean "the light is 1/2 as bright over the right sensor as it is over the left." The actual values for **`tRight`** and **`tLeft`** might be small in a bright room or large in a dark room, but the answer will still be 0.25 if the light is 1/2 as bright over the right sensor. A measurement of 0.5 would mean that the **`tRight`** and **`tLeft`** values are equal. They could both be large, or both be small, but if the result is 0.5, it means the sensors are detecting the same level of brightness.

Here's another trick: subtract 0.5 from the normalized differential shade measurement. That way, the results range from –0.5 to +0.5 instead of 0 to 1, and a measurement of 0 means equal brightness. The result is *a zero-justified* normalized differential shade measurement.

$$zero\ justified\ normalized\ differential\ shade = \frac{tRight}{tRight + tLeft} - 0.5$$

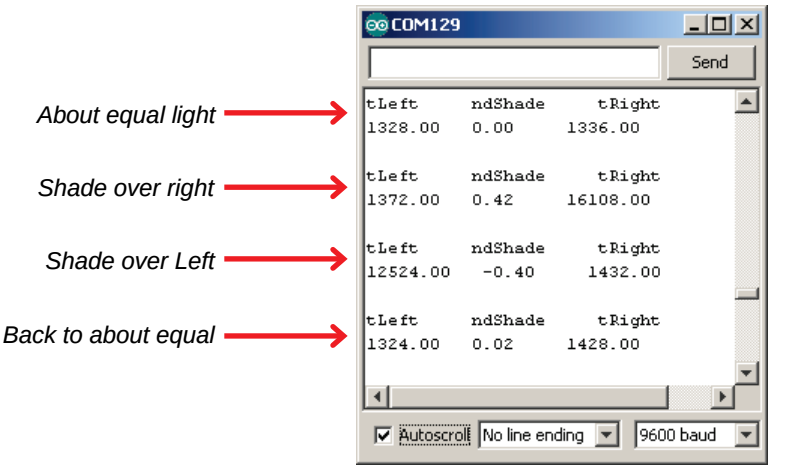
But why do it? The value range –0.5 to +0.5 is great for navigation sketches because the positive and negative values can be used to scale the wheels speeds. Here is how the zero-justified normalized differential shade equation appears in the next sketch:

```
float ndShade;                          // Normalized differential shade
ndShade = tRight / (tLeft + tRight) - 0.5; // Calculate & subtract 0.5
```

The final measurement will be stored in a floating point variable named **ndShade**, so that gets declared first. Then, the next line does the zero-justified normalized differential shade math. The result will be a value in the –0.5 to +0.5 range that represents the fraction of total shade that **tRight** detects, compared to **tLeft**. When **ndShade** is 0, it means **tRight** and **tLeft** are the same values, so the sensors are detecting equally bright light. The closer **ndShade** gets to –0.5, the darker the shade over the right sensor. The closer **ndShade** gets to 0.5 the darker the shade over the left sensor. This will be very useful for navigation. Let's test it first with the Serial Monitor.

Example Sketch: LightSensorValues

This screen capture shows a Serial Monitor example with the LightSensorValues sketch running. With shade over the right sensor, the **ndShade** value is about 0.4. With shade over the left sensor, it's about –0.4.

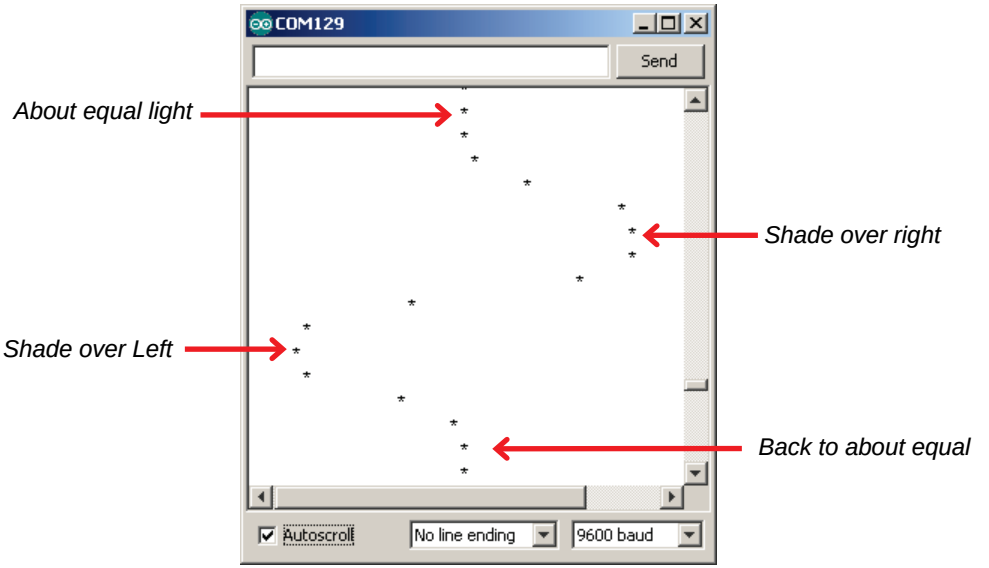


- ✓ Make sure there is no direct sunlight streaming in nearby windows. Indoor lighting is good, but direct sunlight will still flood the sensors.
- ✓ Verify that when you cast shade over the BOE Shield-Bot's left sensor, it results in negative values, with darker shade resulting in larger negative values.
- ✓ Verify that when you cast shade over the BOE Shield-Bot's right sensor, it results in positive values, with darker shade resulting in larger positive values.
- ✓ Verify that when both sensors see about the same level of light or shade, that **ndShade** reports values close to 0.
- ✓ Try casting equal shade over both sensors. Even though the overall light level dropped, the value of **ndShade** should still stay close to zero.

```
/*
 * Robotics with the BOE Shield - LightSensorValues
 * Displays tLeft, ndShade and tRight in the Serial Monitor.
 */

void setup()                              // Built-in initialization block
{
  tone(4, 3000, 1000);                    // Play tone for 1 second
  delay(1000);                            // Delay to finish tone

  Serial.begin(9600);                     // Set data rate to 9600 bps
}

void loop()                               // Main loop auto-repeats
{
  float tLeft = float(rcTime(8));         // Get left light & make float
  float tRight = float(rcTime(6));        // Get right light & make float

  float ndShade;                          // Normalized differential shade
  ndShade = tRight / (tLeft + tRight) - 0.5; // Calculate & subtract 0.5

  // Display heading
  Serial.println("tLeft     ndShade     tRight");

  Serial.print(tLeft);                    // Display tLeft value
  Serial.print("   ");                    // Display spaces
  Serial.print(ndShade);                  // Display ndShade value
  Serial.print("     ");                  // Display more spaces
  Serial.println(tRight);                 // Display tRight value
  Serial.println(' ');                    // Add an extra newline

  delay(1000);                            // 1 second delay
}

long rcTime(int pin)                      // rcTime measures decay at pin
{
  pinMode(pin, OUTPUT);                   // Charge capacitor
  digitalWrite(pin, HIGH);                // ..by setting pin ouput-high
  delay(5);                               // ..for 5 ms
  pinMode(pin, INPUT);                    // Set pin to input
  digitalWrite(pin, LOW);                 // ..with no pullup
  long time  = micros();                  // Mark the time
  while(digitalRead(pin));                // Wait for voltage < threshold
  time = micros() - time;                 // Calculate decay time
  return time;                            // Returns decay time
}
```

Light Measurement Graphic Display

The Serial Monitor screen capture below shows an example of a graphical display of the **ndShade** variable. The asterisk will be in the center of the -0.5 to +0.5 scale if the light or shade is the same over both sensors. If the shade is darker over the BOE Shield-Bot's right sensor, the asterisk will position to the right in the scale. If it's darker over the left, the asterisk will position toward the left. A larger shade/light contrast (like darker shade over one of the sensors) will result in the asterisk positioning further from the center.



- ✓ Load the LightSensorDisplay sketch into the Arduino.
- ✓ Open the Serial Monitor and make sure Autoscroll is checked.

Try casting different levels of shade over each light sensor, and watch how the asterisk in the Serial Monitor responds. Remember that if you cast equal shade over both sensors, the asterisk should still be in the middle; it only indicates which sensor sees more shade if there's a difference between them.

```
/*
 * Robotics with the BOE Shield - LightSensorDisplay
 * Displays a scrolling graph of ndShade.  The asterisk positions ranges
 * from 0 to 40 with 20 (middle of the display) indicating same light on
 * both sides.
 */

void setup()                            // Built-in initialization block
{
```

```
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone
  Serial.begin(9600);                   // Set data rate to 9600 bps
}

void loop()                             // Main loop auto-repeats
{
  float tLeft = float(rcTime(8));       // Get left light & make float
  float tRight = float(rcTime(6));      // Get right light & make float

  float ndShade;                        // Normalized differential shade
  ndShade = tRight / (tLeft+tRight) - 0.5;   // Calculate & subtract 0.5

  for(int i = 0; i<(ndShade * 40) + 20; i++) // Place asterisk in 0 to 40
  {
    Serial.print(' ');                  // Pad (ndShade * 40) + 20 spaces
  }
  Serial.println('*');                  // Print asterisk and newline

  delay(100);                           // 0.1 second delay
}

long rcTime(int pin)                    // rcTime measures decay at pin
{
  pinMode(pin, OUTPUT);                 // Charge capacitor
  digitalWrite(pin, HIGH);              // ..by setting pin ouput-high
  delay(5);                             // ..for 5 ms
  pinMode(pin, INPUT);                  // Set pin to input
  digitalWrite(pin, LOW);               // ..with no pullup
  long time  = micros();                // Mark the time
  while(digitalRead(pin));              // Wait for voltage < threshold
  time = micros() - time;               // Calculate decay time
  return time;                          // Returns decay time
}
```

How LightSensorDisplay Works

The **loop** function starts by taking the two **rcTime** measurements for the left and right light sensors, and stores them in **tLeft** and **tRight**.

```
void loop()                             // Main loop auto-repeats
{
  float tLeft = float(rcTime(8));       // Get left light & make float
  float tRight = float(rcTime(6));      // Get right light & make float
```

After declaring **ndShade** as a floating-point variable, **tLeft** and **tRight** are used in an expression to get that zero-justified normalized differential measurement. The result will be between –0.5 and +0.5, and gets stored in **ndShade**.

```
  float ndShade;                         // Normalized differential shade
```

```
  ndShade = tRight / (tLeft+tRight) - 0.5;   // Calculate & subtract 0.5
```

Next, this **for** loop places the cursor in the right place for printing an asterisk. Take a close look at the **for** loop's condition. It takes **ndShade** and multiples it by 40. It also has to add 20 to the value because if **ndShade** is –0.5, we want that to print with zero leading spaces. So (–0.5 × 40) + 20 = 0. Now, if **ndShade** is 0, we want it to print 20 spaces over: (0 × 40) + 20 = 20. If it's +0.5 we want it to print 40 spaces over: (0.5 × 40) + 20 = 40. Of course, if it's something in between, like 0.25, we have (0.25 × 40) + 20 = 30. So, it'll print half way between center and far right.

```
  for(int i = 0; i<(ndShade * 40) + 20; i++) // Place asterisk in 0 to 40
  {
    Serial.print(' ');                  // Pad (ndShade * 40) + 20 spaces
  }
```

After printing the spaces, a single asterisk prints on the line. Recall that **println** prints and also adds a newline so that the next time through the loop, the asterisk will display on the next line.

```
  Serial.println('*');                  // Print asterisk and newline

  delay(100);                           // 0.1 second delay
}
```

Activity 4: Test a Light-Roaming Routine

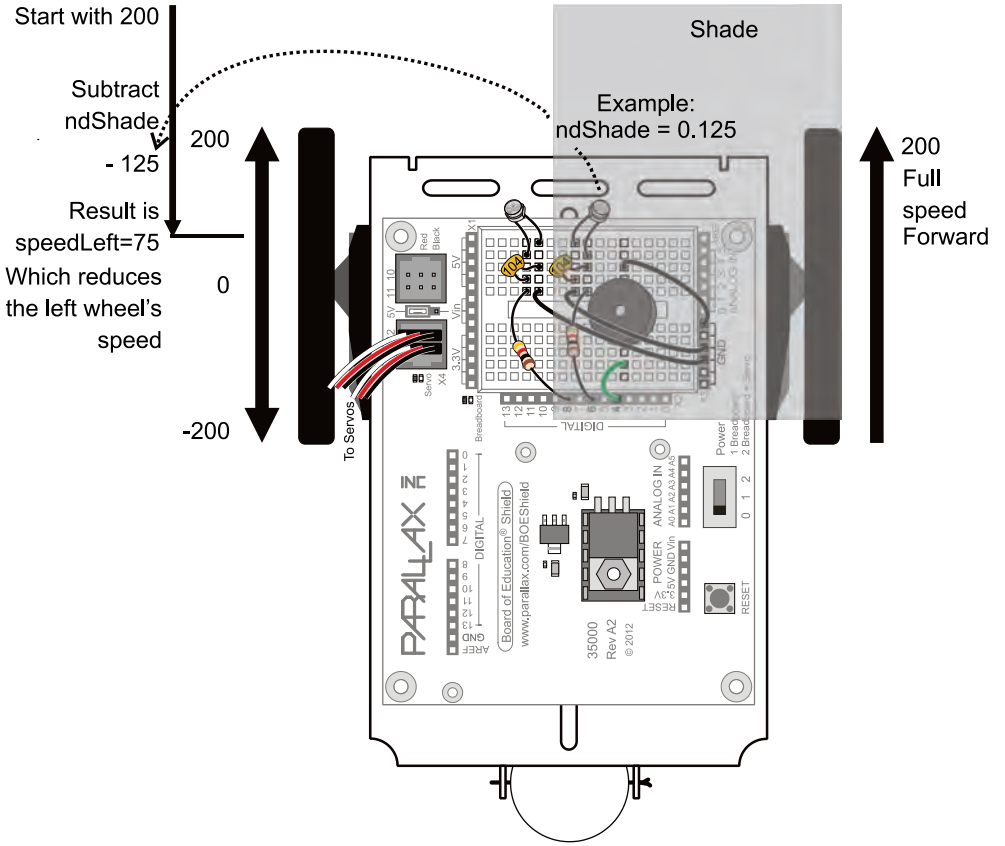
One approach toward making the Boe-Bot roam toward light sources is to make it turn away from shade. You can use the **ndShade** variable to make the BOE Shield-Bot turn a little or a lot when the contrast between the light detected on each side is a little or a lot.

Shady Navigation Decisions

Here is an **if** statement that works well for turning away from shade on the right side of the BOE Shield-Bot. It starts by declaring two **int** variables, **speedLeft** and **speedRight**. They are not declared within the **if…else** block because other blocks in the **loop** function will need to check their values too. Next, **if(ndShade > 0.0)** has a code block that will be executed if shade is detected on the robot's right side, slowing down the left wheel to make the BOE Shield-Bot turn away from the dark. To do this, **ndShade * 1000.0** is subtracted from 200. Before assigning the result to **speedLeft**, **int(200.0–(ndShade×1000.0)** converts the answer from a floating point value back to an integer. We're doing this to make the value compatible with our custom **maneuver** function , which needs an **int** value.

```
int speedLeft, speedRight;                  // Declare speed variables

if (ndShade > 0.0)                          // Shade on right?
{                                           // Slow down left wheel
  speedLeft = int(200.0 - (ndShade * 1000.0));
  speedLeft = constrain(speedLeft, -200, 200);
  speedRight = 200;                         // Full speed right wheel
}
```

This diagram shows an example of how this works when **ndShade** is 0.125. The left wheel slows down because 200 – (0.125×1000) = 75. Since linear speed control is in the 100 to -100 range, it puts the wheel at about ¾ of full speed. Meanwhile, on the other side, **`speedRight`** is set to 200 for full speed forward.



The larger **ndShade** is, the more it subtracts from 200. That's not a problem in this example, but if **ndShade** were 0.45, it would try to store –250 in the **speedLeft** variable. Since the speeds we'll want to pass to the **maneuver** function need to be in the -200 to 200 range, we'll use the Arduino's **constrain** function to prevent **speedLeft** from going out of bounds: **speedLeft = constrain(speedLeft, –200, 200)**.

Here is an **else** statement that works well for turning away from shade on the left. It slows down the right wheel and keeps the left wheel going full speed forward. Notice that it adds **(ndShade*1000)** to 200. Reason being, this is the **else** statement for **if(ndShade > 0.0)**, so it will get used when **ndShade** is equal to or smaller than zero. So, if **ndShade** is –0.125, **speedRight = int(200.0 + (ndShade * 1000.0))** would evaluate to 200 + (–1.25 × 1000) = 200 – 125 = 75. The **constrain** function is used again, to limit **speedRight**.

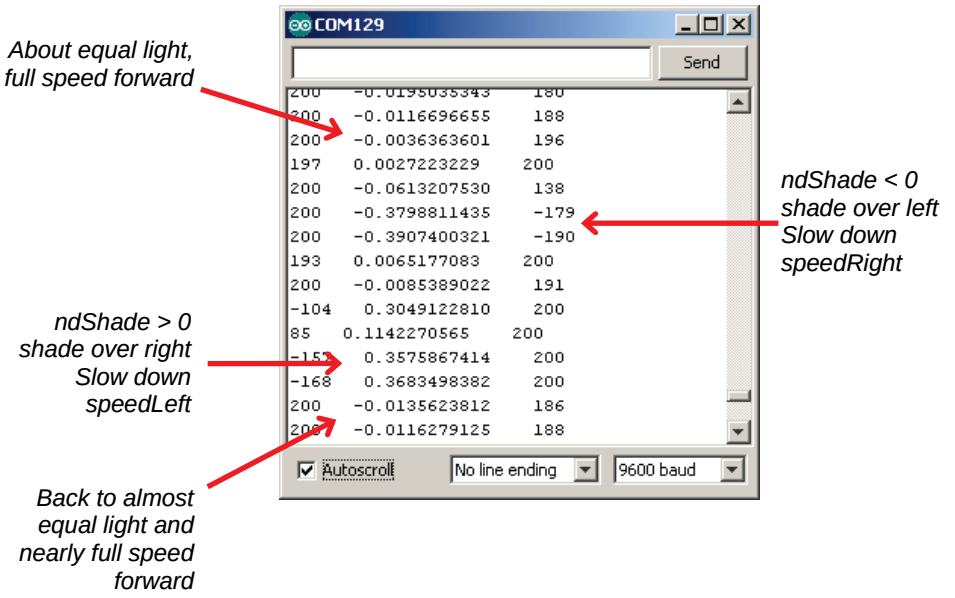
```
  else                                    // Shade on Left?
  {                                       // Slow down right wheel
    speedRight = int(200.0 + (ndShade * 1000.0));
    speedRight = constrain(speedRight, -200, 200);
    speedLeft = 200;                      // Full speed left wheel
  }
```

Test Navigation Decisions with Serial Monitor

Before actually testing out these navigation decisions, it's best to take a look at the variable values with the Serial Monitor. So, instead of a call to the **maneuver** function, first, let's use some **Serial.print** calls to see if we got it right.

```
  Serial.print(speedLeft, DEC);           // Display speedLeft
  Serial.print("   ");                    // Spaces
  Serial.print(ndShade, DEC);             // Display ndShade
  Serial.print("    ");                   // More spaces
  Serial.println(speedRight, DEC);        // Display speedRight

  delay(2000);                            // 1 second delay
}
```

The **print** and **println** calls should result in a display like the one below. It shows the value of **speedLeft** in the left column, **speedRight** in the right column, and **ndShade** between them. Watch it carefully. The side with brighter light will always display 200 for full-speed-forward, and the other will be slowing down with values less than 200—the darker the shade, the smaller the number.



Example Sketch – Light Seeking Display

- ✓ Make sure the power switch is set to 1.
- ✓ Create, save, and run the sketch LightSeekingDisplay.
- ✓ Open the Serial Monitor.
- ✓ Try casting different levels of shade over the BOE Shield-Bot's right light sensor. Does **`speedLeft`** (in the left column) slow down or even go into reverse with lots of shade?
- ✓ Try the same thing with the left light sensor to verify the right wheel slows down.
- ✓ Try casting more shade over both. Again, since the shade is the same for both, **`ndShade`** should stay close to zero, with little if any slowing of the wheels. (Remember, **`speedLeft`** and **`SpeedRight`** would have to drop by 100 before they'll start to slow down.)

```
/*
 * Robotics with the BOE Shield - LightSeekingDisplay
 * Displays speedLeft, ndShade, and speedRight in Serial Monitor. Verifies
 * that wheel speeds respond correctly to left/right light/shade conditions.
 */

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone
```

```
  Serial.begin(9600);                   // Set data rate to 9600 bps
}

void loop()                             // Main loop auto-repeats
{
  float tLeft = float(rcTime(8));       // Get left light & make float
  float tRight = float(rcTime(6));      // Get right light & make float

  float ndShade;                        // Normalized differential shade
  ndShade = tRight / (tLeft+tRight) - 0.5;   // Calculate & subtract 0.5

  int speedLeft, speedRight;            // Declare speed variables

  if (ndShade > 0.0)                    // Shade on right?
  {                                     // Slow down left wheel
    speedLeft = int(200.0 - (ndShade * 1000.0));
    speedLeft = constrain(speedLeft, -200, 200);
    speedRight = 200;                   // Full speed right wheel
  }
  else                                  // Shade on Left?
  {                                     // Slow down right wheel
    speedRight = int(200.0 + (ndShade * 1000.0));
    speedRight = constrain(speedRight, -200, 200);
    speedLeft = 200;                    // Full speed left wheel
  }

  Serial.print(speedLeft, DEC);         // Display speedLeft
  Serial.print("   ");                  // Spaces
  Serial.print(ndShade, DEC);           // Display ndShade
  Serial.print("    ");                 // More spaces
  Serial.println(speedRight, DEC);      // Display speedRight

  delay(1000);                          // 1 second delay
}

long rcTime(int pin)                    // rcTime measures decay at pin
{
  pinMode(pin, OUTPUT);                 // Charge capacitor
  digitalWrite(pin, HIGH);              // ..by setting pin ouput-high
  delay(5);                             // ..for 5 ms
  pinMode(pin, INPUT);                  // Set pin to input
  digitalWrite(pin, LOW);               // ..with no pullup
  long time  = micros();                // Mark the time
  while(digitalRead(pin));              // Wait for voltage < threshold
  time = micros() - time;               // Calculate decay time
  return time;                          // Returns decay time
}
```

Activity 5: Shield-Bot Navigating by Light

At this point, the LightSeekingDisplay sketch needs four things to take it from displaying what it's going to do to actually doing it:

- ✓ Remove the **`Serial.print`** calls.
- ✓ Add servo code.
- ✓ Add the **`maneuver`** function.
- ✓ Add a call to the **`loop`** function to pass **`speedLeft`** and **`speedRight`** to the **`maneuver`** function.

The result is the LightSeekingShieldBot sketch.

- ✓ Create, save, and run the sketch LightSeekingShieldBot.
- ✓ Connect the battery pack, put the BOE Shield-Bot on the floor, and set the power switch to 2.
- ✓ Let the BOE Shield-Bot roam and try casting shadows over its left and right light sensors. It should turn away from the shadow.

```
/*
 * Robotics with the BOE Shield - LightSeekingShieldBot
 * Roams toward light and away from shade.
 */

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12
}

void loop()                             // Main loop auto-repeats
{
  float tLeft = float(rcTime(8));       // Get left light & make float
  float tRight = float(rcTime(6));      // Get right light & make float

  float ndShade;                        // Normalized differential shade
  ndShade = tRight / (tLeft+tRight) - 0.5;   // Calculate & subtract 0.5

  int speedLeft, speedRight;            // Declare speed variables

  if (ndShade > 0.0)                    // Shade on right?
  {                                     // Slow down left wheel
```

```
    speedLeft = int(200.0 - (ndShade * 1000.0));
    speedLeft = constrain(speedLeft, -200, 200);
    speedRight = 200;                   // Full speed right wheel
  }
  else                                  // Shade on Left?
  {                                     // Slow down right wheel
    speedRight = int(200.0 + (ndShade * 1000.0));
    speedRight = constrain(speedRight, -200, 200);
    speedLeft = 200;                    // Full speed left wheel
  }

  maneuver(speedLeft, speedRight, 20);  // Set wheel speeds
}

long rcTime(int pin)                    // rcTime measures decay at pin
{
  pinMode(pin, OUTPUT);                 // Charge capacitor
  digitalWrite(pin, HIGH);              // ..by setting pin ouput-high
  delay(5);                             // ..for 5 ms
  pinMode(pin, INPUT);                  // Set pin to input
  digitalWrite(pin, LOW);               // ..with no pullup
  long time  = micros();                // Mark the time
  while(digitalRead(pin));              // Wait for voltage < threshold
  time = micros() - time;               // Calculate decay time
  return time;                          // Returns decay time
}

// maneuver function
void maneuver(int speedLeft, int speedRight, int msTime)
{
  servoLeft.writeMicroseconds(1500 + speedLeft);   // Left servo speed
  servoRight.writeMicroseconds(1500 - speedRight); // Right servo speed
  if(msTime==-1)                                   // If msTime = -1
  {
    servoLeft.detach();                            // Stop servo signals
    servoRight.detach();
  }
  delay(msTime);                                   // Delay for msTime
}
```

Your Turn – Light/Shade Sensitivity Adjustments

- ✓ If you want more sensitivity to light, change 1000 to a larger value in these two commands:

```
    speedLeft = int(200.0 - (ndShade * 1000.0));
    speedRight = int(200.0 + (ndShade * 1000.0));
```

Want less light sensitivity? Change 1000 to a smaller value.

- ✓ Try it.

Here are several more light-sensing navigation ideas for your BOE Shield-Bot that can be made with adjustments to the **loop** function:

- ✓ To make your BOE Shield-Bot follow shade instead of light, place **ndShade = -ndShade** right before the **if…else** statement. Curious about how or why this works? Check out Project 2 at the end of this chapter.
- ✓ End roaming under a bright light or in a dark cubby by detecting very bright or very dark conditions. Add **tLeft** and **tRight** together, and compare the result to either a really high (dark) threshold value or a really low (bright) threshold value.
- ✓ Make your BOE Shield-Bot function as a light compass by remaining stationary and rotating toward bright light sources.
- ✓ Incorporate whiskers into the roaming toward light activity so that the BOE Shield-Bot can detect and navigate around objects in its way.

Chapter 6 Summary

This chapter focused on using a pair of light sensors to detect bright light and shade for robot navigation. Lots of interesting electronics concepts and programming techniques come into play.

Electronics

- What a phototransistor is, and how to identify its base, emitter and collector
- What wavelengths are in the ultraviolet, visible, and infrared spectrums
- What is meant by ambient light
- What the difference is between a binary sensor and an analog sensor
- What Ohm's Law is, and how to use it to select a resistor to adjust the phototransistor circuit's voltage response
- Using a phototransistor as a simple binary sensor in a voltage output circuit
- What a capacitor is, and how small capacitors for breadboard circuits are labeled in units of picofarads
- Using a phototransistor as an analog sensor in a resistor-capacitor charge-transfer circuit, also called a QT circuit
- What it means when components are connected in series vs. connected in parallel
- What voltage decay is, and how it's used in a resistor-capacitor circuit

Programming

- How to use the Arduino's **analogRead** function to take a voltage measurement from an analog sensor's output
- How a sketch can measure voltage decay time from a resistor-capacitor circuit to take a measurement from an analog sensor

- How to use the Arduino's **`constrain`** function to set upper and lower limits for a variable value

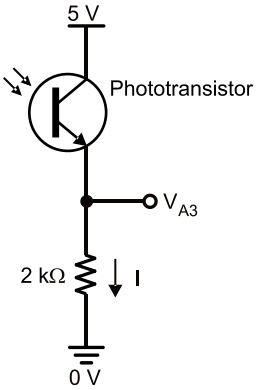
Robotics Skills

- Using a pair of phototransistors for autonomous sensor navigation in response to light level

Engineering Skills

- Subsystem testing of circuits and code routines
- The concept of resolution, in the context of the Arduino reporting a 10-bit value from an analog input
- Using an equation to get a zero-justified normalized differential measurement

Chapter 6 Challenges



Questions

1. What does a transistor regulate?
2. Which phototransistor terminals have leads?
3. How can you use the flat spot on the phototransistor's plastic case to identify its terminals?
4. Which color would the phototransistor be more sensitive to: red or green?
5. How does V_{A3} in the circuit above respond if the light gets brighter?
6. What does the phototransistor in the circuit above do that causes V_{A3} to increase or decrease?
7. How can the circuit above be modified to make it more sensitive to light?

8. What happens when the voltage applied to an I/O pin that has been set to input is above or below the threshold voltage?
9. If the amount of charge a capacitor stores decreases, what happens to the voltage at its terminals?

Exercises

1. Solve for V_{A3} if I = 1 mA in the circuit above.
2. Calculate the current through the resistor if V_{A3} in the circuit above is 4.5 V.
3. Calculate the value of a capacitor that has been stamped 105.
4. Write an **`rcTime`** statement that measures decay time with pin 7 and stores the result in a variable named **`tDecay`**.
5. Calculate what the **`ndShade`** measurement would be if the Arduino measures decay values of 1001 on both sides.
6. Write a **`for`** loop that displays fifty equal sign characters in the Serial Monitor.

Projects

1. In Activity 1 the circuit, along with the HaltUnderBrightLight sketch, made the BOE Shield-Bot stop under a light at the end of the course. What if you will only have a limited time at a course before a competition, and you don't know the lighting conditions in advance? You might need to calibrate your BOE Shield-Bot on site. A sketch that makes the piezospeaker beep repeatedly when the BOE Shield-Bot detects bright light and stay quiet when it detects ambient light could be useful for this task. Write and test a sketch to do this with the circuit in Activity 1.
2. Develop an application that makes the BOE Shield-Bot roam and search for darkness instead of light. This application should utilize the charge transfer circuits from Building the Photosensitive Eyes on page 189.
3. Develop an application that makes the BOE Shield-Bot roam toward a bright incandescent desk lamp in a room where the only other light sources are fluorescent ceiling lights. The BOE Shield-Bot should be able to roam toward the desk lamp and play a tone when it's under it. This application should use the charge transfer circuits from Building the Photosensitive Eyes on page 189.

Question Solutions

1. The amount of current it allows to pass into its collector and out through its base.
2. The phototransistor's collector and emitter terminals are connected to leads.
3. The lead that's closer to the flat spot is the emitter. The lead that's further away from the flat spot is the collector.
4. The wavelength of red is closer to the wavelength of infrared, so it should be more sensitive to red.
5. V_{A3} increases with more light.

6. The phototransistor supplies the resistor with more or less current.
7. Change the 2 kΩ resistor to a higher value.
8. If the applied voltage is above the threshold voltage, the input register bit for that pin stores a 1. If it's below threshold voltage, the input register bit stores a 0.
9. The voltage decreases.

Exercise Solutions

1. V = I × R = 0.001 A × 2000 Ω = 2 V.
2. V = I × R → I = V ÷ R = 4.5 ÷ 2000 = 0.00225 A = 2.25 mA.
3. 105 → 10 with 5 zeros appended and multiplied by 1 pF. 1,000,000 × 1 pF = (1 × 106) × (1 × 10–12) F = 1 × 10–6 F = 1 µF.
4. It would be **`long tDecay = rcTime(7);`**
5. **`ndShade = tRight / (tLeft+tRight)`** - 0.5 = 1001 ÷ (1001 + 1001) – 0.5 = 0.5 – 0.5 = 0.
6. Solution:

```
for(int i = 1; i<=50; i++)  // Repeat 50 times
{
  Serial.print('=');        // one = char each time through
}
```

Project Solutions

1. This is a modified version of HaltUnderBrightLight

```
/*
 * Robotics with the BOE Shield - Chapter 6, Project 1
 * Chirp when light is above threshold.  Will require updating value of
 * threshold & retesting under bright light to get to the right value.
 */

void setup()                          // Built-in initialization block
{
  tone(4, 3000, 1000);                // Play tone for 1 second
  delay(1000);                        // Delay to finish tone
}

void loop()                           // Main loop auto-repeats
{
  if(volts(A3) > 3.5)                  // If A3 voltage greater than 3.5
  {
    tone(4, 4000, 50);                // Start chirping
    delay(100);
  }
}
```

```
float volts(int adPin)                // Measures volts at adPin
{                                     // Returns floating point voltage
 return float(analogRead(adPin)) * 5.0 / 1024.0;
}
```

2. The solution for this one is to make a copy of LightSeekingShieldBot, and add one command to the **loop** function: **ndShade = -ndShade**. Add it right before the **if…else** statement. Then, instead of indicating shade to turn away from, it indicates bright light to turn away from. Another approach would be to use an **ndLight** calculation that equals **tLeft / (tLeft + tRight)**. You would have to search/replace **ndShade** with **ndLight** in the sketch.

   Use LightSensorValues to determine a threshold. It's best to take **tLeft + tRight**. You can use the LightSensorValues to test this. Start with a value that's 1/4 of the way from reaching the bright light level. So if **tLeft** + **tRight** = 4000 and 400 for bright light, use the value 400 + ¼ × (4000 – 400) = 400 + 900 = 1300. Don't just use 1300, it's just an example; figure out the value for your conditions.

   Next, add an **if** statement similar to the one from HaltUnderBrightLight to the main **loop** of LightSeekingShieldBot. Careful though, HaltUnderBrightLight uses the greater than (**>**) operator because it's using a voltage output circuit. We need the less than (**<**) operator for the QT circuit because smaller values mean brighter light. We also need to express the threshold as a floating point value, like 1300.0. Here's an example:

```
  // Add this if condition to stop under the bright lamp.
  if((tRight + tLeft) < 1300.0)   // tLeft+tRight < 1300?
  {
    servoLeft.detach();           // Stop servo signals
    servoRight.detach();
  }
```

3. Here's a modified version of LightSeekingShieldBot that will do the trick. Remember, you'll still have to calibrate it to your lighting conditions.

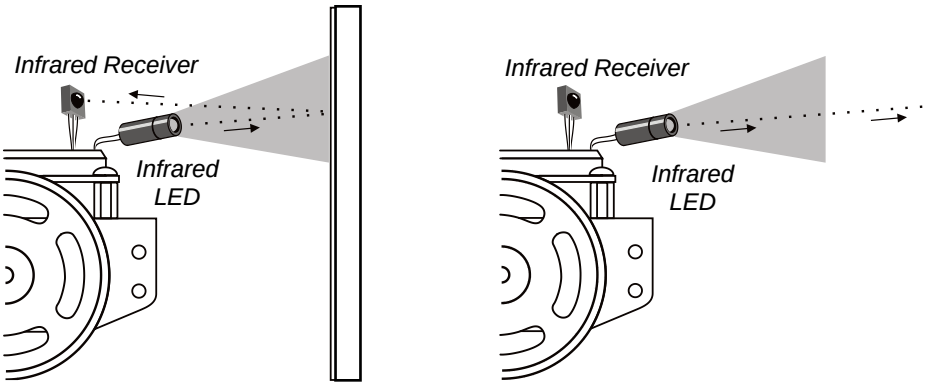
```
/*
 * Robotics with the BOE Shield - Chapter 6, Project 3
 * Roams toward light and away from shade.
 */
```

```
#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12
}

void loop()                             // Main loop auto-repeats
{
  float tLeft = float(rcTime(8));       // Get left light & make float
  float tRight = float(rcTime(6));      // Get right light & make float

  // Add this if condition to stop under the bright lamp.
  if((tRight + tLeft) < 1300.0)         // If A3 voltage greater than 2
  {
    servoLeft.detach();                 // Stop servo signals
    servoRight.detach();
  }

  float ndShade;                        // Normalized differential shade
  ndShade = tRight / (tLeft+tRight) - 0.5; // Calculate & subtract 0.5

  int speedLeft, speedRight;             // Declare speed variables

  if (ndShade > 0.0)                     // Shade on right?
  {                                      // Slow down left wheel
    speedLeft = int(200.0 - (ndShade * 1000.0));
    speedLeft = constrain(speedLeft, -200, 200);
    speedRight = 200;                    // Full speed right wheel
  }
  else                                   // Shade on Left?
  {                                      // Slow down right wheel
    speedRight = int(200.0 + (ndShade * 1000.0));
    speedRight = constrain(speedRight, -200, 200);
    speedLeft = 200;                     // Full speed left wheel
  }

  maneuver(speedLeft, speedRight, 20);   // Set wheel speeds
}

long rcTime(int pin)                     // rcTime measures decay at pin
{
  pinMode(pin, OUTPUT);                  // Charge capacitor
  digitalWrite(pin, HIGH);               // ..by setting pin ouput-high
  delay(5);                              // ..for 5 ms
  pinMode(pin, INPUT);                   // Set pin to input
  digitalWrite(pin, LOW);                // ..with no pullup
```

```
  long time  = micros();                // Mark the time
  while(digitalRead(pin));              // Wait for voltage < threshold
  time = micros() - time;               // Calculate decay time
  return time;                          // Returns decay time
}

// maneuver function
void maneuver(int speedLeft, int speedRight, int msTime)
{
  servoLeft.writeMicroseconds(1500 + speedLeft);   // Left servo speed
  servoRight.writeMicroseconds(1500 - speedRight); // Right servo speed
  if(msTime==-1)                                   // if msTime = -1
  {
    servoLeft.detach();                 // Stop servo signals
    servoRight.detach();
  }
  delay(msTime);                        // Delay for msTime
}
```

Chapter 7. Navigating with Infrared Headlights

The BOE Shield-Bot can already use whiskers to get around, but it only detects obstacles when it bumps into them. Wouldn't it be convenient if the BOE Shield-Bot could just "see" objects and then decide what to do about them? Well, that's what it can do with infrared headlights and eyes like the ones shown below. Each headlight is an infrared LED inside a tube that directs the light forward, just like a flashlight. Each eye is an infrared receiver that sends the Arduino high/low signals to indicate whether it detects the infrared LED's light reflected off an object.

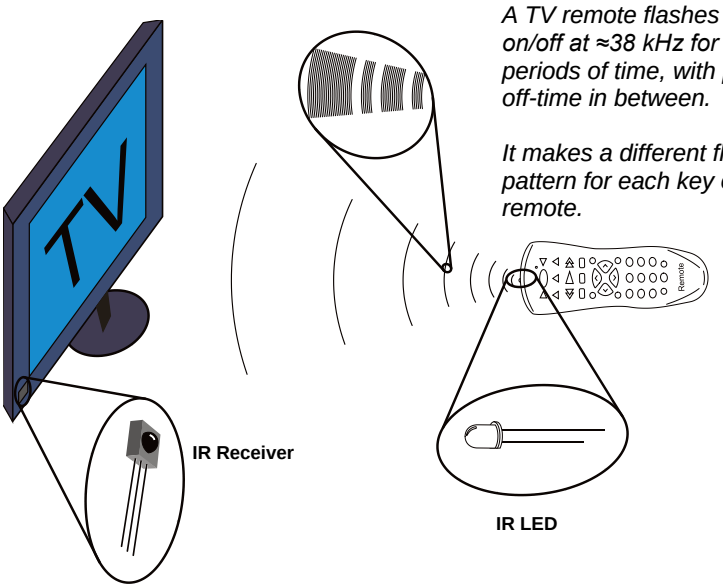


Infrared Light Signals

Infrared is abbreviated IR, and it is light the human eye cannot detect (for a color image of the visible light spectrum, see http://learn.parallax.com/lightspectrum). The IR LEDs introduced in this chapter emit infrared light, just like the red LEDs we've been using emit visible light.

The infrared receivers in this chapter detect infrared light, similar to the phototransistors in the last chapter. But, there's a difference—these infrared receivers are not just detecting ambient light, but they are designed to detect infrared light flashing on and off very quickly.

The infrared LED that the BOE Shield-Bot will use as a tiny headlight is actually the same kind you can find in just about any TV remote. The TV remote flashes the IR LED to send messages to your TV. The microcontroller in your TV picks up those messages with an infrared receiver like the one your BOE Shield-Bot will use.

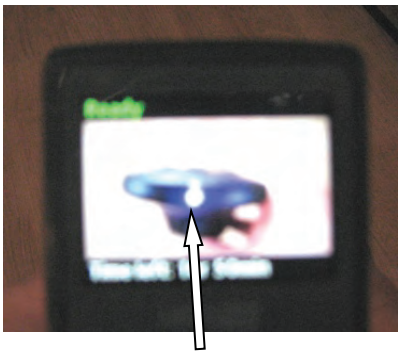


The TV remote sends messages by flashing the IR LED very fast, at a rate of about 38 kHz (about 38,000 times per second). The IR receiver only responds to infrared if it's flashing at this rate. This prevents infrared from sources like the sun and incandescent lights from being misinterpreted as messages from the remote. So, to send signals that the IR receiver can detect, your Arduino will have to flash the IR LED on/off at 38 kHz.

IR Interference: Some fluorescent lights do generate signals that can be detected by the IR receivers. These lights can cause problems for your BOE Shield-Bot's infrared headlights. One of the things you will do in this chapter is develop an infrared interference "sniffer" that you can use to test the fluorescent lights near your BOE Shield-Bot courses.

The light sensors inside many digital cameras, including some cell phones and webcams, can all detect infrared light. By looking through a digital camera, we can "see" if an infrared LED is on or off. These photos show an example with a digital camera and a TV remote. When you press and hold a button on the remote and point the IR LED into the digital camera's lens, it displays the infrared LED as a flashing, bright white light.

With a button pressed and held, the IR LED doesn't look any different.



Through a digital camera display, the IR LED appears as a bright white light.

The pixel sensors inside the digital camera detect red, green, and blue light levels, and the processor adds up those levels to determine each pixel's color and brightness. Regardless of whether a pixel sensor detects red, green, or blue, it detects infrared. Since all three pixel color sensors also detect infrared, the digital camera display mixes all the colors together, which results in white.

Infra means below, so infrared means below red.
The name refers to the fact that the frequency of infrared light waves is less than the frequency of red light waves. The wavelength our IR LED transmits is 980 nanometers (abbreviated nm), and that's the same wavelength our IR receiver detects. This wavelength is in the near-infrared range. The far-infrared range is 2000 to 10,000 nm, and certain wavelengths in this range are used for night-vision goggles and IR temperature sensing.

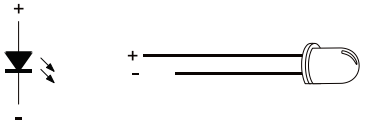
Activity 1: Build and Test the Object Detectors

In this activity, you will build and test infrared object detectors for the BOE Shield-Bot.

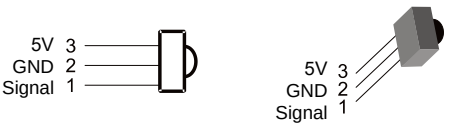
Parts List

(2) IR receivers
(2) IR LEDs (clear case)
(2) IR LED shield assemblies
(2) Resistors, 220 Ω (red-red-brown)
(2) Resistors, 2 kΩ (red-black-red)
(misc) Jumper wires

- ✓ Gather the parts in the Parts List, using the drawings below to help identify the infrared receivers, LEDs, and shield assembly parts.



Infrared LED



Infrared Receiver

Infrared LED Shield Assembly

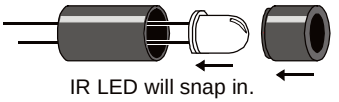
✓ Check the figure below to make sure you have selected infrared LEDs and not phototransistors. The infrared LED has a taller and more rounded plastic dome, and is shown on the right side of this drawing.

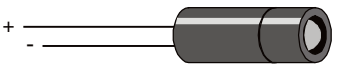
Assemble the IR Headlights

✓ Insert the infrared LED into the LED standoff base (the larger of the two pieces) as shown below. The standoff is shaped to fit the flat side of the LED.

✓ Make sure the IR LED snaps into the LED standoff.

✓ Slip the short tube over the IR LED's clear plastic case. The ring on one end of the tube should fit right into the LED standoff with a little twist.



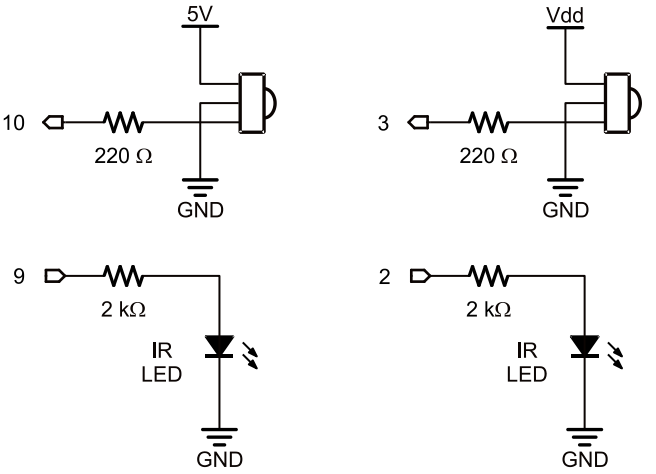


IR Object Detection Circuit

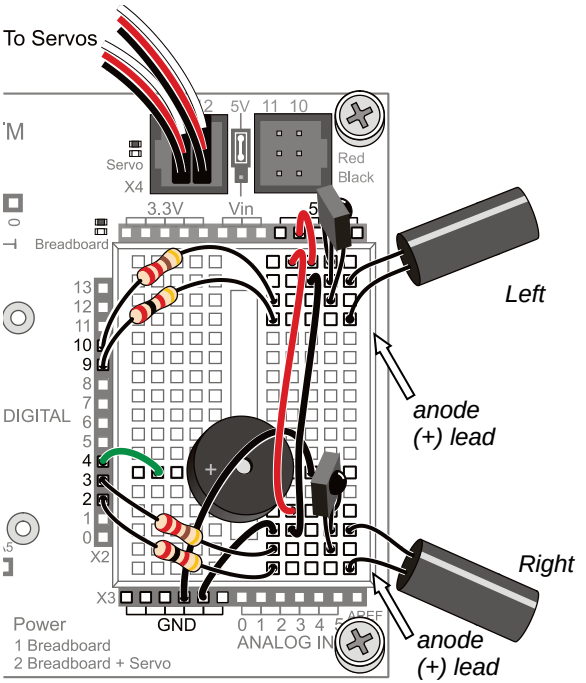
The next figures show the IR object detection schematic and wiring diagram. One IR object detector (IR LED and receiver pair) is mounted on each corner of the breadboard closest to the very front of the BOE Shield-Bot.

- ✓ Disconnect the power and programming cables.
- ✓ Build the circuit in the schematic below, using the wiring diagram as a reference for parts placement.

Note that the anode lead of each IR LED connects to a 2 kΩ resistor. The cathode lead plugs into the same breadboard row as an IR detector's center pin, and that row is connected to GND with a jumper wire.



Watch your IR LED anodes and cathodes!
The anode lead is the longer lead on an IR LED by convention. The cathode lead is shorter and mounted in the plastic case closer to its flat spot. These are the same conventions as the red LEDs we have been using.



Object Detection Test Code

Your BOE Shield-Bot's infrared receivers are designed to detect infrared light (in the 980 nanometer range) flashing at a rate near 38 kHz. To make the IR LED turn on/off at that rate, we can use the familiar **tone** function that makes the speaker beep at the beginning of each sketch.

Infrared detection takes three steps:

1. Flash the IR LED on/off at 38 kHz.
2. Delay for a millisecond or more to give the IR receiver time to send a low signal in response to sensing 38 kHz IR light reflecting off an object.
3. Check the state of the IR receiver for either a high signal (no IR detected), or a low signal (IR detected).

Here is an example of the three steps applied to the left IR LED (pin 9) and IR receiver (pin 10).

```
tone(9, 38000, 8);                // IRLED 38 kHz for at least 1 ms
delay(1);                         // Wait 1 ms
int ir = digitalRead(10);         // IR receiver -> ir variable
```

The tone actually lasts about 1.1 ms. Even though we would expect **`tone(9, 38000, 8)`** to generate a 38 kHz signal for 8 ms, the signal actually only lasts for about 1.1 ms as of Arduino software v1.0. Given the name **`tone`**, the function may have been designed for and tested with audible tones. If played on a speaker, a 38 kHz tone will not be audible. It's in the ultrasonic range, meaning it's pitch is so high that it's above the audible range for humans, which is about 20 Hz to 20 kHz.

The **`tone`** function generates a square wave that repeats its high/low cycle 38000 times per second. Through experimentation with Arduino software v1.0, the author discovered that a call to **`tone`** with a duration of 8 ms resulted in a 38 kHz square wave that lasted about 1.1 ms.

Remember that the **`tone`** function does its processing in the background. Since the IR receiver needs a few tenths of a millisecond to respond to the 38 kHz signal, **`delay(1)`** prevents the **`ir = digitalRead(10)`** call from copying the IR receiver's output until it's ready.

The next sketch defines a function named **`irDetect`** with three parameters: one to specify the IR LED pin, one to specify the IR receiver pin, and one to set the frequency for flashing the IR LED.

```
int irDetect(int irLedPin, int irReceiverPin, long frequency)
```

In the **`loop`** function you'll see a call to **`irDetect`**:

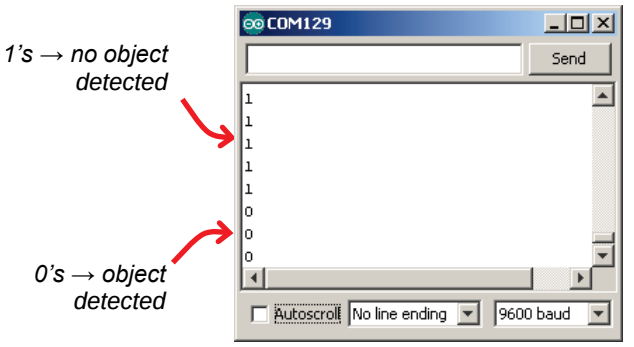
```
int irLeft = irDetect(9, 10, 38000);       // Check for object
```

This call passes 9 to the **`irLedPin`** parameter, 10 to **`irReceiverPin`**, and 38000 to the ***`frequency`*** parameter. The function performs those three steps for infrared detection and returns 1 if no object is detected, or 0 if an object is detected. That return value gets stored in **`irLeft`**.

Example Sketch: TestLeftIr

This sketch only tests the BOE Shield-Bot's left IR detector. This helps simplify trouble-shooting because you are focusing on only one of the two circuits. This is yet another example of subsystem testing. After the subsystems check out, we can move to system integration. But first, you've got to make sure to test and correct any wiring or code entry errors that might have crept in.

- ✓ Reconnect the battery pack to the Arduino.
- ✓ Set the 3-position switch to position 1.
- ✓ Create, save, and run the sketch the sketch TestLeftIr.




```
/*
 * Robotics with the BOE Shield - TestLeftIR
 * Display 1 if the left IR detector does not detect an object,
 * or 0 if it does.
 */

void setup()                              // Built-in initialization block
{
  tone(4, 3000, 1000);                    // Play tone for 1 second
  delay(1000);                            // Delay to finish tone

  pinMode(10, INPUT);  pinMode(9, OUTPUT);   // Left IR LED & Receiver

  Serial.begin(9600);                     // Set data rate to 9600 bps
}

void loop()                               // Main loop auto-repeats
{
  int irLeft = irDetect(9, 10, 38000);  // Check for object

  Serial.println(irLeft);                 // Display 1/0 no detect/detect

  delay(100);                             // 0.1 second delay
}

// IR Object Detection Function

int irDetect(int irLedPin, int irReceiverPin, long frequency)
{
  tone(irLedPin, frequency, 8);           // IRLED 38 kHz at least 1 ms
  delay(1);                               // Wait 1 ms
  int ir = digitalRead(irReceiverPin);  // IR receiver -> ir variable
  delay(1);                               // Down time before recheck
  return ir;                              // Return 1 no detect, 0 detect
}
```

✓ Leave the BOE Shield-Bot connected to its programming cable, and open the Serial Monitor when the sketch is done loading.

- ✓ Place an object, such as your hand or a sheet of paper, about an inch (2 to 3 cm) from the left IR object detector.
- ✓ Verify that the Serial Monitor displays a 0 when you place an object in front of the IR object detector, and a 1 when you remove the object.
- ✓ If the Serial Monitor displays the expected values, go ahead and test the right IR Object Detector (below). If not, go to the Troubleshooting section for help.

Your Turn – Test the Right IR Object Detector

Modifying the sketch to test the right IR object detector is a matter of replacing **`irLeft`** with **`irRight`**, and passing the correct pin numbers to the **`irDetect`** parameters to test the other circuit. Here's a checklist of the changes:

- ✓ Save the sketch TestLeftIr as TestRightIr.
- ✓ Change **`pinMode(10, INPUT); pinMode(9, OUTPUT)`** to **`pinMode(3, INPUT); pinMode(2, OUTPUT)`**.
- ✓ Change **`int irLeft = irDetect(9, 10, 38000)`** to **`int irRight = irDetect(2, 3, 38000)`**.
- ✓ Change **`Serial.println(irLeft)`** to **`Serial.println(irRight)`**.
- ✓ Repeat the testing steps in this activity for the BOE Shield-Bot's right IR object detector.
- ✓ If necessary, trouble-shoot any circuit or code entry errors.

Activity 2: Field Testing

In this activity, you will build and test indicator LEDs that will tell you if an object is detected without the help of the Serial Monitor. This is handy if you are not near your computer, and you need to troubleshoot your IR detector circuits.

You will also write a sketch to "sniff" for infrared interference from fluorescent lights. Some fluorescent lights send signals that resemble the signal sent by your infrared LEDs. The device inside a fluorescent light fixture that controls voltage for the lamp is called the *ballast*. Some ballasts operate in the same frequency range of your IR detector, causing the lamp to emit a 38.5 kHz infrared signal. When using IR object detection for navigation, ballast interference can cause some bizarre BOE Shield-Bot behavior!

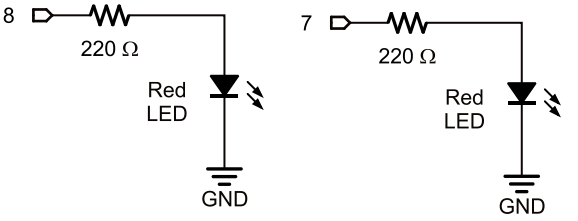
Adding LED Indicator Circuits

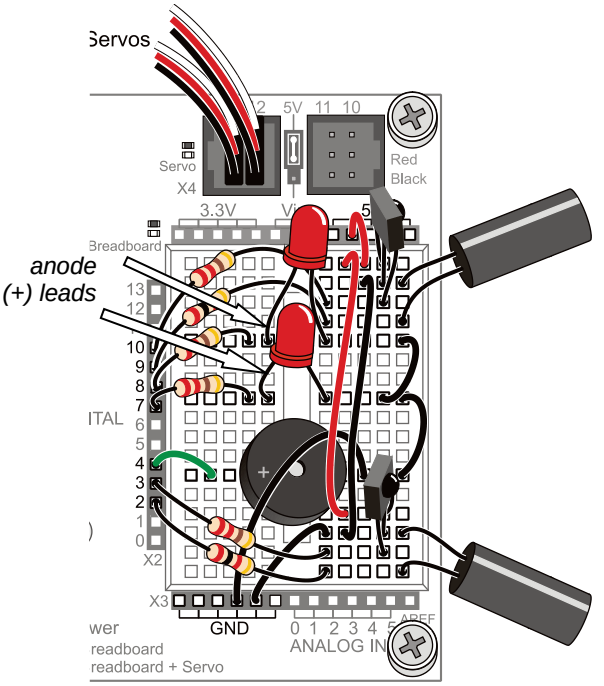
LED indicator circuits are similar to the ones you used with the whiskers. Make sure to be careful about your cathodes and anodes when you connect them.

Parts List

(2) Red LEDs
(2) Resistors, 220 Ω (red-red-brown)
(misc) Jumper wires

- ✓ Disconnect the programming and power cables.
- ✓ Add the red LED circuits shown in the schematic to the BOE Shield, using the wiring diagram to help with parts placement.





Testing the System

There are quite a few components involved in this system, and this increases the likelihood of a wiring error. That's why it's important to have a test sketch that shows you what the infrared detectors are sensing. Use this sketch to verify that all the circuits are working before unplugging the BOE Shield-Bot from its programming cable.

Example Sketch – TestBothIrAndIndicators

- ✓ Reconnect the programming and power cables, and set the Power switch to 1.
- ✓ Create, save, and run the sketch TestBothIrAndIndicators.
- ✓ Verify that the speaker makes a clear, audible tone after the sketch loads.
- ✓ Use the Serial Monitor to verify that the Arduino still receives a zero from each IR detector when an object is placed in front of it.
- ✓ Verify that when you place an object in front of the left IR detector, the left (pin 8) red LED lights up.
- ✓ Repeat that test for the right IR detector and pin 7 LED.

```
/*
 * Robotics with the BOE Shield – TestBothIrAndIndicators
 * Test both IR detection circuits with the Serial Monitor.  Displays 1 if
 * the left IR detector does not detect an object, or 0 if it does.
 * Also displays IR detector states with indicator LEDs.
 */

void setup()                              // Built-in initialization block
{
  tone(4, 3000, 1000);                    // Play tone for 1 second
  delay(1000);                                 // Delay to finish tone

  pinMode(10, INPUT);  pinMode(9, OUTPUT);   // Left IR LED & Receiver
  pinMode(3, INPUT);  pinMode(2, OUTPUT);    // Right IR LED & Receiver
  pinMode(8, OUTPUT); pinMode(7, OUTPUT);    // Indicator LEDs

  Serial.begin(9600);                     // Set data rate to 9600 bps
}

void loop()                               // Main loop auto-repeats
{
  int irLeft = irDetect(9, 10, 38000);  // Check for object on left
  int irRight = irDetect(2, 3, 38000);  // Check for object on right

  digitalWrite(8, !irLeft);               // LED states opposite of IR
  digitalWrite(7, !irRight);

  Serial.print(irLeft);                   // Display 1/0 no detect/detect
  Serial.print("  ");                     // Display 1/0 no detect/detect
  Serial.println(irRight);                // Display 1/0 no detect/detect
```

```
  delay(100);                           // 0.1 second delay
}

// IR Object Detection Function

int irDetect(int irLedPin, int irReceiverPin, long frequency)
{
  tone(irLedPin, frequency, 8);         // IRLED 38 kHz for at least 1 ms
  delay(1);                             // Wait 1 ms
  int ir = digitalRead(irReceiverPin);  // IR receiver -> ir variable
  delay(1);                             // Down time before recheck
  return ir;                            // Return 1 no detect, 0 detect
}
```

Inside TestBothIrAndIndicators

After the two calls to the **irDetect** function, **irLeft** and **irRight** each store a 1 if an object is not detected, or 0 if an object is detected. The sketch could have used an **if…else** statement to turn the indicator lights on/off depending on the value of **irLeft** and **irRight**. But, there are other ways!

This approach uses the values of **irLeft** and **irRight** directly. There's only one catch: when (for example) **irLeft** stores 0, we want its red indicator LED to turn on, and if it stores 1, we want its LED to turn off. Since 1 makes the indicator LED turn on and 0 turns it off, using **digitalWrite(8, irLeft)** would give us the opposite of what we want. So, the sketch uses the not operator (**!**) to invert the value that **irLeft** stores. Now, **digitalWrite(8, !irLeft)** inverts the value of **IrLeft** so that when it stores a zero, the LED turns on, and when it stores 1, the LED turns off. The same technique is applied for the right IR detector and indicator LED.

The Not (!) Operator inverts all the binary values in a variable. There's more going on with **digitalWrite(8, !irLeft)** here than meets the eye. You're probably used to passing the **digitalWrite** function's ***value*** parameter either **HIGH** (constant for 1) to turn the light on, or **LOW** (constant for 0) to turn the light off. When you use variables, the **digitalWrite** function uses the variable's rightmost binary digit: 1 to turn the light on, or 0 to turn the light off. So, when **irLeft** is 0 (object detected) **!irLeft** changes its binary value 00000000 to 11111111. It has a 1 in the rightmost digit, so the light turns on. When **irLeft** is 1, it's really binary 00000001. The **!irLeft** statement results in binary 11111110. Rightmost digit is 0, so the light turns off.

Your Turn – Remote Testing and Range Testing

With the indicator LEDs working, you can now unplug the BOE Shield-Bot from its programming cable and test detection with a variety of objects. Since certain objects reflect infrared better than others, the IR object detectors will be able to see some objects further away, and others only when very close.

- ✓ Make sure the battery pack is connected to your Arduino and the BOE Shield's Power switch is set to 1.
- ✓ Unplug your BOE Shield-Bot from the programming cable, and test again with objects in the area where you plan to make it navigate.
- ✓ Test the detection range with different colored objects. What colors, materials, and surface textures can it detect only at closest range? What can it detect from farther away?

Sniffing for IR Interference

You might have found that your BOE Shield-Bot said it detected something even though nothing was in range. That may mean a nearby light is generating some IR light at a frequency close to 38.5 kHz. It might also mean that direct sunlight streaming through a window is causing false detections. If you try to have a BOE Shield-Bot contest or demonstration near one of these light sources, your infrared systems could end up performing very poorly! So, before any public demo, make sure to check the prospective navigation area with this IR interference "sniffer" sketch ahead of time.

The concept behind this sketch is simple: don't transmit any IR through the IR LEDs, just monitor to see if any IR is detected. If IR is detected, sound the alarm using the piezospeaker.

You can use a handheld remote for just about any piece of equipment to generate IR interference. TVs, VCRs, CD/DVD players, and projectors all use the same type of IR detectors you have on your BOE Shield-Bot right now. So, the remotes you use to control these devices all use the same kind of IR LED that's on your BOE Shield-Bot to transmit messages to your TV, VCR, CD/DVD player, etc. All you'll have to do to generate IR interference is point the remote at your BOE Shield-Bot and repeatedly press/release one of the remote's buttons.

Example Sketch – IrInterferenceSniffer

With this sketch, your BOE Shield-Bot should play a tone, turn on its indicator LEDs, and display a warning in the Serial Monitor any time it detects infrared. Again, since it's not transmitting any IR, it means the 38 kHz infrared has to be coming from an outside source.

- ✓ Create, save, and run the sketch IrInterferenceSniffer.
- ✓ Test to make sure the BOE Shield-Bot sounds the alarm when it detects IR interference. If you are in a classroom, you can do this with a separate BOE Shield-Bot that's running TestBothIrAndIndicators. Just point its IR LEDs into the IrInterferenceSniffer bot's IR receivers. If you don't have a second BOE Shield-Bot, just use a handheld remote for a TV, VCR, CD/DVD player, or projector. Simply point the remote at the BOE Shield-Bot and repeatedly press and release

one of its buttons. If the BOE Shield-Bot responds by sounding the alarm, you know your IR interference sniffer is working.

```
/*
 * Robotics with the BOE Shield - IrInterferenceSniffer
 * Test for external sources of IR interference.  If IR interference is
 * detected: Serial Monitor displays warning, piezospeaker plays alarm,
 * and indicator lights flash.
 */

void setup()                              // Built-in initialization block
{
  tone(4, 3000, 1000);                    // Play tone for 1 second
  delay(1000);                            // Delay to finish tone

  pinMode(10, INPUT);                     // Left IR Receiver
  pinMode(3, INPUT);                      // Right IR Receiver
  pinMode(8, OUTPUT);                     // Left indicator LED
  pinMode(7, OUTPUT);                     // Right indicator LED

  Serial.begin(9600);                     // Set data rate to 9600 bps
}

void loop()                               // Main loop auto-repeats
{
  int irLeft = digitalRead(10);           // Check for IR on left
  int irRight = digitalRead(3);           // Check for IR on right

  if((irLeft == 0) || (irRight == 0))     // If left OR right detects
  {
    Serial.println("IR interference!!!");    // Display warning
    for(int i = 0; i < 5; i++)            // Repeat 5 times
    {
      digitalWrite(7, HIGH);              // Turn indicator LEDs on
      digitalWrite(8, HIGH);
      tone(4, 4000, 10);                  // Sound alarm tone
      delay(20);                          // 10 ms tone, 10 between tones
      digitalWrite(7, LOW);               // Turn indicator LEDs off
      digitalWrite(8, LOW);
    }
  }
}
```

Your Turn – Testing for Fluorescent Lights that Interfere

- ✓ Disconnect your BOE Shield-Bot from its programming cable, and point it at any fluorescent light near where you plan to operate it.
- ✓ Especially if you get frequent alarms, turn off that fluorescent light before trying to use IR object detection under it.

- ✓ If the source turns out to be sunlight streaming in through a window, close the blinds and retest, or move your obstacle course to a location that's well away from the window.

Always use this IrInterferenceSniffer to make sure that any area where you are using the BOE Shield-Bot is free of infrared interference.

Activity 3: Detection Range Adjustments

You may have noticed that brighter car headlights (or a brighter flashlight) can be used to see objects that are further away when it's dark. By making the BOE Shield-Bot's infrared LED headlights brighter, you can also increase its detection range. A smaller resistor allows more current to flow through an LED. More current through an LED is what causes it to glow more brightly. In this activity, you will examine the effect of different resistance values with both the red and infrared LEDs.

Parts List:

You will need some extra parts for this activity:

(2) Resistors, 470 Ω (yellow-violet-brown)
(2) Resistors, 220 Ω (red-red-brown)
(2) Resistors, 1 kΩ (brown-black-red)
(2) Resistors, 4.7 kΩ (yellow-violet-red)

Series Resistance and LED Brightness

First, let's use one of the red LEDs to see the difference that a resistor makes in how brightly an LED glows. All we need to test the LED is a sketch that sends a high signal to the LED.

Example Sketch – P1LedHigh

- ✓ Create, save, and run the sketch LeftLedOn.
- ✓ Run the sketch and verify that the LED in the circuit connected to P8 emits light.

```
// Robotics with the BOE Shield - LeftLedOn
// Turn on left LED for brightness testing

void setup()                              // Built-in initialization block
{
  tone(4, 3000, 1000);                    // Play tone for 1 second
  delay(1000);                            // Delay to finish tone

  pinMode(8, OUTPUT);                     // Left indicator LED
  digitalWrite(8, HIGH);
}

void loop()                               // Main loop auto-repeats
{
}
```

Testing LED Brightness with Different Resistors

Remember to disconnect power and the programming cable before you make changes to a circuit. Remember also that the same sketch will run again when you reconnect power, so you can pick up right where you left off with each test.

- ✓ Replace the 220 Ω resistor that goes from pin 8 to the right LED's cathode with a 470 Ω resistor. Note now how brightly the LED glows.
- ✓ Repeat for a 1 kΩ resistor.
- ✓ Repeat once more with a 4.7 kΩ resistor.
- ✓ Replace the 4.7 kΩ resistor with the 220 Ω resistor before moving on to the next portion of this activity.
- ✓ Explain in your own words the relationship between LED brightness and series resistance.

Series Resistance and IR Detection Range

We now know that less series resistance will make an LED glow more brightly. A reasonable hypothesis would be that brighter IR LEDs can make it possible to detect objects that are further away.

- ✓ Re-open and run TestBothIrAndIndicators.
- ✓ Verify that both LED indicator circuits are working properly before continuing.

Your Turn – Testing IR LED Range

- ✓ With a ruler, measure the furthest distance from the IR LED that a sheet of paper can be detected when using 2 kΩ resistors, and record your data in the next table.

- ✓ Replace the 2 kΩ resistors that connect pin 2 and pin 9 to the IR LED anodes with 4.7 kΩ resistors.
- ✓ Determine the furthest distance at which the same sheet of paper is detected, and record your data.
- ✓ Repeat with 1 kΩ resistors, 470 Ω resistors, and 220 Ω resistors. (For the smaller resistor values, they may end up making the detectors so sensitive that they see the table surface in front of your robot. If that happens, set the robot on the edge of the table with the IR detectors pointing off the end and try the distance measurements again.)

Detection Distances vs. Resistance	
IRELD Series Resistance (Ω)	**Maximum Detection Distance**
4700	
2000	
1000	
470	
220	

- ✓ Before moving on to the next activity, restore your IR object detectors to their original circuit, with 2 kΩ resistors in series with each IR LED.
- ✓ Also, before moving on, make sure to test this last change with TestBothIrAndIndicators to verify that both IR object detectors are working properly.

Activity 4: Object Detection and Avoidance

An interesting thing about these IR detectors is that their outputs are just like the whiskers. When no object is detected, the output is high; when an object is detected, the output is low. In this activity, the example sketch RoamingWithWhiskers is modified so that it works with the IR detectors; all it takes is few simple modifications. Here are the steps:

1. Save the sketch RoamingWithWhiskers as RoamingWithIr
2. Add the **`irDetect`** function.

```
int irDetect(int irLedPin, int irReceiverPin, long frequency)
{
  tone(irLedPin, frequency, 8);
  delay(1);
  int ir = digitalRead(irReceiverPin);
  delay(1);
```

```
  return ir;
}
```

3. Replace these **`digitalRead`** calls:

```
  byte wLeft = digitalRead(5);
  byte wRight = digitalRead(7);
```

4. …with these calls to **`irDetect`**:

```
  int irLeft = irDetect(9, 10, 38000);
  int irRight = irDetect(2, 3, 38000);
```

5. Replace all instances of **`wLeft`** with **`irLeft`** and **`wRight`** with **`irRight`**.
6. Update the **`/*…*/`** and **`//`** comments.

Example Sketch – RoamingWithIr

- ✓ Open RoamingWithWhiskers.
- ✓ Save it as RoamingWithIr.
- ✓ Modify it so that it matches the sketch below.
- ✓ Save the sketch and run it on the Arduino.
- ✓ Disconnect the BOE Shield-Bot from its programming cable.
- ✓ Reconnect the battery pack and move the 3-position switch to position 2.
- ✓ Place your BOE Shield-Bot somewhere where it can roam and avoid obstacles.
- ✓ Verify that it behaves like RoamingWithWhiskers (aside from the fact that there's no contact required).

```
/*
 * Robotics with the BOE Shield - RoamingWithIr
 * Adaptation of RoamingWithWhiskers with IR object detection instead of
 * contact switches.
 */

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

void setup()                            // Built-in initialization block
{
  pinMode(10, INPUT);  pinMode(9, OUTPUT);   // Left IR LED & Receiver
  pinMode(3, INPUT);  pinMode(2, OUTPUT);    // Right IR LED & Receiver
```

```
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12
}

void loop()                             // Main loop auto-repeats
{
  int irLeft = irDetect(9, 10, 38000);  // Check for object on left
  int irRight = irDetect(2, 3, 38000);  // Check for object on right

  if((irLeft == 0) && (irRight == 0))   // If both sides detect
  {
    backward(1000);                     // Back up 1 second
    turnLeft(800);                      // Turn left about 120 degrees
  }
  else if(irLeft == 0)                  // If only left side detects
  {
    backward(1000);                     // Back up 1 second
    turnRight(400);                     // Turn right about 60 degrees
  }
  else if(irRight == 0)                 // If only right side detects
  {
    backward(1000);                     // Back up 1 second
    turnLeft(400);                      // Turn left about 60 degrees
  }
  else                                  // Otherwise, no IR detected
  {
    forward(20);                        // Forward 1/50 of a second
  }
}

int irDetect(int irLedPin, int irReceiverPin, long frequency)
{
  tone(irLedPin, frequency, 8);         // IRLED 38 kHz for at least 1 ms
  delay(1);                             // Wait 1 ms
  int ir = digitalRead(irReceiverPin);  // IR receiver -> ir variable
  delay(1);                             // Down time before recheck
  return ir;                            // Return 1 no detect, 0 detect
}

void forward(int time)                  // Forward function
{
  servoLeft.writeMicroseconds(1700);    // Left wheel counterclockwise
  servoRight.writeMicroseconds(1300);   // Right wheel clockwise
  delay(time);                          // Maneuver for time ms
}

void turnLeft(int time)                 // Left turn function
{
  servoLeft.writeMicroseconds(1300);    // Left wheel clockwise
  servoRight.writeMicroseconds(1300);   // Right wheel clockwise
  delay(time);                          // Maneuver for time ms
```

```
}

void turnRight(int time)                // Right turn function
{
  servoLeft.writeMicroseconds(1700);    // Left wheel counterclockwise
  servoRight.writeMicroseconds(1700);   // Right wheel counterclockwise
  delay(time);                          // Maneuver for time ms
}

void backward(int time)                 // Backward function
{
  servoLeft.writeMicroseconds(1300);    // Left wheel clockwise
  servoRight.writeMicroseconds(1700);   // Right wheel counterclockwise
  delay(time);                          // Maneuver for time ms
}
```

Activity 5: High-performance IR Navigation

The style of pre-programmed maneuvers from the last activity were fine for whiskers, but are unnecessarily slow when using the IR detectors. With whiskers, the BOE Shield-Bot had to make contact and then back up to navigate around obstacles. With infrared, your BOE Shield-Bot will detect most obstacles before it runs into them, and can just find a clear path around the obstacle.

Increase the Sampling Rate to Avoid Collisions

You can reduce all your maneuver durations to 20 ms, which means your sketch will check and recheck for objects almost 50 times per second. (It's actually a little slower, more like 40 times per second, because the code execution and IR detection all takes time too.) As your BOE Shield-Bot navigates, it will execute a series of small turns to avoid an obstacle before it ever runs into it. With that approach, it never turns further than it has to, and it can neatly find its way around obstacles and successfully navigate more complex courses. After experimenting with this next sketch, you'll likely agree that it's a much better way for the BOE Shield-Bot to roam.

Example Sketch – FastIrRoaming

- ✓ Create, save, and run the sketch FastIrRoaming, then test it with the same obstacles you used from the previous activity.

```
/*
 * Robotics with the BOE Shield - FastIrRoaming
 * Adaptation of RoamingWithWhiskers with IR object detection instead of
 * contact switches
 */

#include <Servo.h>                      // Include servo library
```

```
Servo servoLeft;                      // Declare left and right servos
Servo servoRight;

void setup()                          // Built-in initialization block
{
  pinMode(10, INPUT);  pinMode(9, OUTPUT);   // Left IR LED & Receiver
  pinMode(3, INPUT);  pinMode(2, OUTPUT);    // Right IR LED & Receiver

  tone(4, 3000, 1000);                // Play tone for 1 second
  delay(1000);                        // Delay to finish tone

  servoLeft.attach(13);               // Attach left signal to pin 13
  servoRight.attach(12);              // Attach right signal to pin 12
}

void loop()                           // Main loop auto-repeats
{

  int irLeft = irDetect(9, 10, 38000);  // Check for object on left
  int irRight = irDetect(2, 3, 38000);  // Check for object on right

  if((irLeft == 0) && (irRight == 0))   // If both sides detect
  {
    maneuver(-200, -200, 20);           // Backward 20 milliseconds
  }
  else if(irLeft == 0)                  // If only left side detects
  {
    maneuver(200, -200, 20);            // Right for 20 ms
  }
  else if(irRight == 0)                 // If only right side detects
  {
    maneuver(-200, 200, 20);            // Left for 20 ms
  }
  else                                  // Otherwise, no IR detects
  {
    maneuver(200, 200, 20);             // Forward 20 ms
  }
}

int irDetect(int irLedPin, int irReceiverPin, long frequency)
{
  tone(irLedPin, frequency, 8);         // IRLED 38 kHz for at least 1 ms
  delay(1);                             // Wait 1 ms
  int ir = digitalRead(irReceiverPin);  // IR receiver -> ir variable
  delay(1);                             // Down time before recheck
  return ir;                            // Return 1 no detect, 0 detect
}

void maneuver(int speedLeft, int speedRight, int msTime)
{
  // speedLeft, speedRight ranges: Backward Linear Stop Linear Forward
  //                                -200    -100....0.....100   200
  servoLeft.writeMicroseconds(1500 + speedLeft);   // Left servo speed
```

```
  servoRight.writeMicroseconds(1500 - speedRight); // Right servo speed
  if(msTime==-1)                                   // If msTime = -1
  {
    servoLeft.detach();                            // Stop servo signals
    servoRight.detach();
  }
  delay(msTime);                                   // Delay for msTime
}
```

How FastIrRoaming Works

This sketch uses the **`maneuver`** function from the TestManeuverFunction sketch. The **`maneuver`** function expects three parameters: **`speedLeft`**, **`speedRight`**, and **`msTime`**. Recall that both speed parameters use 200 for full speed forward, -200 for full speed backward, and values between -100 and +100 for linear speed control. Also, remember **`msTime`** values are 20, so each maneuver executes for 20 ms before returning to the **`loop`** function.

```
void loop()                              // Main loop auto-repeats
{
  int irLeft = irDetect(9, 10, 38000);  // Check for object on left
  int irRight = irDetect(2, 3, 38000);  // Check for object on right

  if((irLeft == 0) && (irRight == 0))   // If both sides detect
  {
    maneuver(-200, -200, 20);           // Backward 20 milliseconds
  }
  else if(irLeft == 0)                  // If only left side detects
  {
    maneuver(200, -200, 20);            // Right for 20 ms
  }
    else if(irRight == 0)               // If only right side detects
  {
    maneuver(-200, 200, 20);            // Left for 20 ms
  }
  else                                  // Otherwise, no IR detects
  {
    maneuver(200, 200, 20);             // Backward 20 ms
  }
}
```

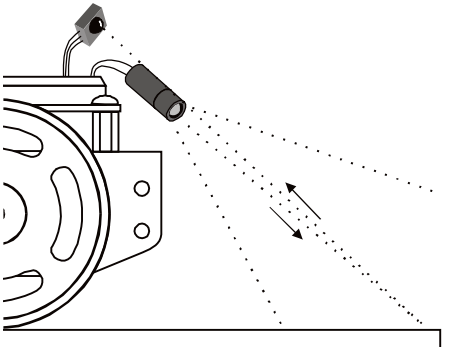
Your Turn

- ✓ Save FastIrRoaming as FastIrRoamingYourTurn.
- ✓ Add code to make the LEDs indicate that the BOE Shield-Bot has detected an object.
- ✓ Try modifying the values that **`speedLeft`** and **`speedRight`** are set to so that the BOE Shield-Bot does everything at half speed. (Remember that 200 and -200 are overkill, so try 50 for half speed forward and –50 for half speed backward).

Activity 6: Drop-off Detector

Up until now, your robot has mainly been programmed to take evasive maneuvers when an object is detected. There are also applications where the BOE Shield-Bot must take evasive action when an object is not detected. For example, if the BOE Shield-Bot is roaming on a table, its IR detectors might be looking down at the table surface. The sketch should make it continue forward so long as both IR detectors can "see" the surface of the table.

- ✓ Disconnect power and programming cable.
- ✓ Point your IR object detectors downward and outward as shown below.



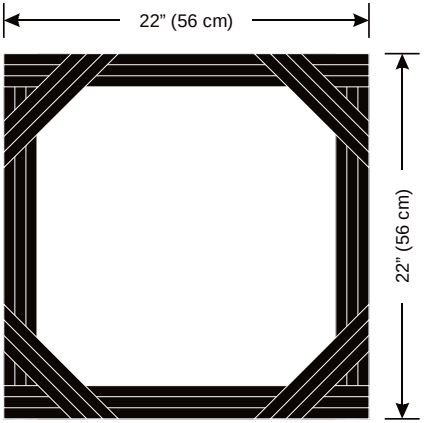
Recommended Materials:

(1) Roll of black vinyl electrical tape, ¾″ (19 mm) wide, or black tempera paint and brush.
(1) Sheet of white poster board, 22 x 28 in (56 x 71 cm).

Simulating a Drop-Off with Poster Board

A sheet of white poster board with a border made of electrical tape or black tempera paint makes for a handy way to simulate the drop-off presented by a table edge, with much less risk to your BOE Shield-Bot.

- ✓ Build a course like one shown below. For electrical tape, make a small test patch on a piece of paper first to make sure the infrared detectors cannot see a reflection from it. If it does not reflect infrared, use three strips edge to edge on the poster board with no paper visible between the strips.



- ✓ Run the sketch IrInterferenceSniffer to make sure that nearby fluorescent lighting will not interfere with your BOE Shield-Bot's IR detectors.
- ✓ Use TestBothIrAndIndicators to make sure that the BOE Shield-Bot detects the poster board but does not detect the electrical tape or paint.

If the BOE Shield-Bot still "sees" the electrical tape too clearly:

- ✓ Try adjusting the IR detectors and LEDs downward at shallower angles.
- ✓ Try a different brand of vinyl electrical tape.
- ✓ Try replacing the 2 kΩ resistors with 4.7 kΩ (yellow-violet-red) resistors to make the BOE Shield-Bot more nearsighted.
- ✓ Adjust the **`tone`** command with different **`frequency`** arguments. Here are some arguments that will make the BOE Shield-Bot more nearsighted: 39000, 40000, 41000.

If the Boe Shield-Bot seems nearsighted:

If you are using older IR LEDs, the BOE Shield-Bot might actually be having problems with being too nearsighted**.** Here are some remedies that will increase the BOE Shield-Bot's sensitivity to objects and make it more far sighted:

- ✓ Try replacing the 2 kΩ (red-black-red) resistors with 1 kΩ (red-black-brown) or even 470 Ω (yellow-violet-brown) resistors in series with the IR LEDs instead of 2 kΩ.
- ✓ Try pointing the IR LEDs and detectors downward at a steeper angle so that the bot is looking at the surface right in front of it.

If you want to try a tabletop:

- ✓ Try an electrical tape or paint course first!
- ✓ Remember to follow the same steps you followed before running the BOE Shield-Bot in the electrical tape or paint delimited course!
- ✓ Always be ready to pick your BOE Shield-Bot up from above as it approaches the edge of the table it's navigating. If the BOE Shield-Bot tries to drive off the edge, pick it up before it takes the plunge. Otherwise, your Shield-Bot might become a Not-Bot!
- ✓ Your BOE Shield-Bot may detect you if you are standing in its line of sight. Its current sketch has no way to differentiate you from the table below it, so it might try to continue forward and off the edge of the table. So, stay out of its detector's line of sight as you spot.

Example Sketch: AvoidTableEdge

For the most part, programming your BOE Shield-Bot to navigate around a tabletop without going over the edge is a matter of adjusting the **`if...else if...else`** statements from FastIrRoaming. First of all, instead of backing up, it will need to go forward 20 ms at a time when it sees objects with both detectors. It will also need to turn toward objects instead of away from them, and it will need to turn for more than 20 ms when it sees the drop-off. 375 ms turns seem to work well, but it will be up to you to adjust that value for best performance.

- ✓ Open FastIrNavigation and save it as AvoidTableEdge.
- ✓ Modify the sketch so that it matches the Example Sketch. Pay close attention to the conditions and **`maneuver`** calls in the **`loop`** function. The condition that used to go forward for 20 ms now backs up for 250 ms. Likewise, the condition that used to back up now goes forward for 20 ms. Also, the condition that used to call for a 20 ms right turn now calls for a 375 ms left turn, and the condition that used to call for a 20 ms left turn now calls for a 375 ms right turn.
- ✓ Test the sketch on your electrical tape or paint delimited course.
- ✓ If you decide to try a tabletop, remember to follow the testing and spotting tips discussed earlier.

```
/*
 * Robotics with the BOE Shield - AvoidTableEdge
 * Adaptation of FastIrRoaming for table edge avoidance
 */

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;
```

```
void setup()                              // Built-in initialization block
{
  pinMode(10, INPUT);  pinMode(9, OUTPUT);   // Left IR LED & Receiver
  pinMode(3, INPUT);  pinMode(2, OUTPUT);    // Right IR LED & Receiver

  tone(4, 3000, 1000);                    // Play tone for 1 second
  delay(1000);                            // Delay to finish tone

  servoLeft.attach(13);                   // Attach left signal to pin 13
  servoRight.attach(12);                  // Attach right signal to pin 12
}

void loop()                               // Main loop auto-repeats
{

  int irLeft = irDetect(9, 10, 38000);  // Check for object on left
  int irRight = irDetect(2, 3, 38000);  // Check for object on right

  if((irLeft == 0) && (irRight == 0))   // Both sides see table surface
  {
    maneuver(200, 200, 20);             // Forward 20 milliseconds
  }
  else if(irLeft == 0)                  // Left OK, drop-off on right
  {
    maneuver(-200, 200, 375);           // Left for 375 ms
  }
  else if(irRight == 0)                 // Right OK, drop-off on left
  {
    maneuver(200, -200, 375);           // Right for 375 ms
  }
  else                                  // Drop-off straight ahead
  {
    maneuver(-200, -200, 250);          // Backward 250 ms before retry
  }
}

int irDetect(int irLedPin, int irReceiverPin, long frequency)
{
  tone(irLedPin, frequency, 8);         // IRLED 38 kHz for at least 1 ms
  delay(1);                             // Wait 1 ms
  int ir = digitalRead(irReceiverPin);  // IR receiver -> ir variable
  delay(1);                             // Down time before recheck
  return ir;                            // Return 1 no detect, 0 detect
}

void maneuver(int speedLeft, int speedRight, int msTime)
{
  // speedLeft, speedRight ranges: Backward Linear Stop Linear Forward
  //                                -200    -100....0....100    200
  servoLeft.writeMicroseconds(1500 + speedLeft);   // Left servo speed
  servoRight.writeMicroseconds(1500 - speedRight); // Right servo speed
  if(msTime==-1)                                   // If msTime = -1
  {
    servoLeft.detach();                            // Stop servo signals
```

```
    servoRight.detach();
  }
  delay(msTime);                                  // Delay for msTime
}
```

How AvoidTableEdge Works

Since AvoidTableEdge is just FastIrRoaming with a modified **if…else if…else** statement in its **loop** function, let's look at the two statements side by side.

```
// From FastIrRoaming

  if((irLeft == 0) && (irRight == 0))
  {
    maneuver(-200, -200, 20);
  }
  else if(irLeft == 0)
  {
    maneuver(200, -200, 20);
  }
  else if(irRight == 0)
  {
    maneuver(-200, 200, 20);
  }
  else
  {
    maneuver(200, 200, 20);
  }
```

```
//From AvoidTableEdge

  if((irLeft == 0) && (irRight == 0))
  {
    maneuver(200, 200, 20);
  }
  else if(irLeft == 0)
  {
    maneuver(-200, 200, 375);
  }
  else if(irRight == 0)
  {
    maneuver(200, -200, 375);
  }
  else
  {
    maneuver(-200, -200, 250);
  }
}
```

In response to **if((irLeft == 0) && (irRight == 0))**, FastIrRoaming backs up with **maneuver(-200, -200, 20)** because both IR detectors see an obstacle. In contrast, AvoidTableEdge goes forward with **maneuver(200, 200, 20)** because both IR detectors see the table, which means it's safe to move forward for another 20 ms.

In response to else **if(irLeft == 0)**, FastIrRoaming turns right for 20 ms, taking a step toward avoiding an obstacle on the left with **maneuver(200, -200, 20)**. AvoidTableEdge instead turns away from a drop-off that must be on its right. That's because the first **if** statement where both IR detectors could see the table was not true. If it was, the **if…else if…else** statement would not have made it to this **else if** condition. It means

the left IR detector does see the table, but the right one does not. So the code makes the robot turn left for 0.375 seconds with **`maneuver(-200, 200, 375)`**. This should make it avoid a drop-off that must have been detected on the right.

In response to **`else if (irRight == 0)`**, FastIrRoaming turns left for 20 ms, taking an incremental step toward avoiding an obstacle on the right. At this point in the **`if…else if…else`** statement, we know that the right IR detector does see an object, but the left one does not. Again, that's because, it would have handled the condition where both IR detectors see the table with the first if statement. So, the left IR detector does not see the object and turns right for 0.375 seconds with **`maneuver(200, -200, 375)`**.

Your Turn

The 375 ms turns to avoid the table edge can be adjusted for different applications. For example, if the BOE Shield-Bot is supposed to hug the edge of the table, smaller turns might be useful. In a contest where the BOE Shield-Bot is supposed to push objects out of an area, a larger turn (but not too large) would be better so that it zigzags back and forth across the table.

You can modify the code to make shallower turns by using a smaller **`msTime`** parameter value in the **`maneuver`** function calls. For example, if you change the 375 in **`maneuver(-200, 200, 375)`** to 300, it will make shallower left turns. If you change it to 450, it will make sharper left turns.

- ✓ Modify AvoidTableEdge so that it closely follows the edge of the simulated drop-off course. Adjust the **`msTime`** parameter values in calls to **`maneuver`** to make the BOE Shield-Bot execute smaller turns when it sees the drop-off. How small can you make the turn before it tries to fall off?
- ✓ Experiment with pivoting away from the table edge to make the BOE Shield-Bot roam inside the perimeter instead of following the edge.

Chapter 7 Summary

This chapter focused on using a pair of infrared LED emitters and receivers to broadcast and detect reflections of 38.5 kHz modulated infrared light. Using this sensor system for robot navigation used these concepts and skills:

Electronics

- What an infrared LED is, and how to identify its anode and cathode
- What a modulated infrared receiver is, and how to use it in a microcontroller circuit
- What effect resistor values have on LEDs when connected in series

Programming

- How to use the familiar Arduino **`tone`** function for a new purpose: to modulate an infrared light signal, instead of control the pitch of a sound
- How to repurpose an existing sketch from one sensor system for use with a different sensor system, so long as both sensors have a similar output signal
- How to repurpose an existing sketch designed for a specific behavior (avoiding obstacles) to a new behavior (detecting a drop-off)

Robotics Skills

- Using a pair of infrared emitters and receivers to detect objects for non-contact autonomous sensor navigation

Engineering Skills

- Using human-visible indicators (red LEDs) to broadcast the state of sensors detecting human-invisible conditions (reflected modulated infrared light)
- More practice with subsystem testing and troubleshooting

Chapter 7 Challenges

Questions

1. What is the frequency of the signal sent by **`tone(9, 38000, 8)`**? How long does this call send the signal for? What I/O pin does the IR LED circuit have to be connected to in order to broadcast this signal?
2. What has to happen after the tone command to find out if the IR receiver detects reflected infrared?
3. What does it mean if the IR detector sends a low signal? What does it mean when the detector sends a high signal?
4. What happens if you change the value of a resistor in series with a red LED? What happens if you change the value of a resistor in series with an infrared LED??

Exercises

1. Modify a line of code in IrInterferenceSniffer so that it only monitors one of the IR detectors.
2. Modify AvoidTableEdge so that when it makes the BOE Shield-Bot back up, it also turns a little so that it doesn't get stuck going straight back at the same drop-off.

Projects

1. Design a BOE Shield-Bot application that sits still until you wave your hand in front of it, then it starts roaming.
2. Design a BOE Shield-Bot application that slowly rotates in place until it detects an object. As soon as it detects an object, it locks onto and chases the object. This is a classic 'SumoBot' behavior.
3. Design a BOE Shield-Bot application that roams, but if it detects infrared interference, it sounds the alarm briefly, and then continues roaming. This alarm should be different from the low battery alarm.

Question Solutions

1. 38 kHz is the frequency of the signal. Although the 8 in **`tone`** call's ***`ms`*** parameter calls for a signal that lasts for 8 ms, the book said that testing indicated it lasted for 1.1 ms. The IR LED must be connected to the digital pin 8 socket for the tone signal to make it flash on/off at 38 kHz.
2. The code has to pause for 1 ms with **`delay(1)`** and then store the IR value by copying whatever **`digitalRead`** returns to a variable; for example, **`delay(1)`** followed by **`irLeft = digitalRead(10)`**.
3. A low signal means that 38 kHz IR was detected, which means that the 38 kHz flashing light the IR LED transmitted must have reflected off an object in its path. In other words: object detected. A high signal means no 38 kHz IR was detected, so, there wasn't any object to reflect the IR LED's light. In other words: object not detected.
4. For both red and infrared LEDs, a smaller resistor will cause the LED to glow more brightly. A larger resistor results in dimmer LEDs. In terms of results, brighter IR LEDs make it possible to detect objects that are farther away. But it can also have annoying side effects, like seeing the reflection of the surface it is rolling on and misinterpreting it as an obstacle.

Exercise Solutions

1. Let's remove the left IR detector from the code. Comment out **`int irLeft = digitalRead(10)`** by placing **`//`** to its left. Then, change **`if((irLeft == 0) || (irRight == 0))`** to **`if(irRight == 0)`**.
2. Remember that linear wheel speed control is in the -100 to 100 range. So -75 should be a little slower than full speed backward. With that in mind, try changing **`maneuver(-200, -200, 250)`** to **`maneuver(-75, -200, 250)`**.

Project Solutions

1. One approach would be to add this code at the end of the **setup** function in FastIrRoaming. It will wait until an object is detected before allowing the code to move on to the roaming code in the **loop** function.

```
  int irLeft = 1;                         // Declare IR variables and
  int irRight = 1;                        // initialize both to 1

  while((irLeft == 1) && (irRight == 1))  // Wait for detection
  {
    irLeft = irDetect(9, 10, 38000);      // Check for object on left
    irRight = irDetect(2, 3, 38000);      // Check for object on right
  }
```

2. Rotating in place and waiting to see an object is almost the same as project 1. The trick is to call the **maneuver** function to make it rotate slowly. Remember that the **maneuver** function's **msTime** parameter just dictates low long the **maneuver** routine prevents another maneuver from making the BOE Shield-Bot do something different. You can instead use it to start a maneuver in a set-it-and forget-it fashion. Then, let the code continue to the loop that monitors the IR detectors until it detects an object.

```
  int irLeft = 1;                         // Declare IR variables and
  int irRight = 1;                        // initialize both to 1

  while((irLeft == 1) && (irRight == 1))  // Wait for detection
  {
    irLeft = irDetect(9, 10, 38000);      // Check for object on left
    irRight = irDetect(2, 3, 38000);      // Check for object on right
  }
```

Now, to chase after an object, the **if…else if…else** statement needs to be updated so that it goes forward when it sees an object with both sensors. It should also turn toward an object instead of away from it. Here is a modified **loop** function.

```
void loop()                               // Main loop auto-repeats
{

  int irLeft = irDetect(9, 10, 38000);    // Check for object on left
  int irRight = irDetect(2, 3, 38000);    // Check for object on right

  if((irLeft == 0) && (irRight == 0))     // If both sides detect
  {
    maneuver(200, 200, 20);               // Forward 20 ms, chase object
  }
```

```
  else if(irLeft == 0)                  // If only left side detects
  {
    maneuver(-200, 200, 20);            // Left toward object 20 ms
  }
    else if(irRight == 0)               // If only right side detects
  {
    maneuver(200, -200, 20);            // Right toward object 20 ms
  }
  else                                  // Otherwise, no IR detects
  {
    maneuver(200, 200, 20);             // Forward 20 ms
  }
}
```

The example above goes forward for 20 ms if it does not see the object. Optionally, it could spin in place until it sees an object and then chase after it again. For this, you could place the code that spins and waits for a detection in a function. Then, your code could call that function from the end of the **setup** function to wait until it sees something, and also from the **else** statement in the **loop** function for when it can't see the object any more.

3. The key to solving this problem is to combine FastIrRoaming and the **tone** portion of IrInterferenceSniffer in a single sketch. This example also turns on the lights while it sounds the alarm. One note, the compiler in the Arduino software won't let you declare a variable twice in the same code block. So, in the **loop** function, the first time the **irLeft** and **irRight** variables are used, they are declared with **int**. The second time within the loop, they are just reused for detection (without using **int** to declare them) because they have already been declared. Modified **setup** and **loop** functions are shown here:

```
void setup()                            // Built-in initialization block
{
  pinMode(10, INPUT);  pinMode(9, OUTPUT);   // Left IR LED & Receiver
  pinMode(3, INPUT);  pinMode(2, OUTPUT);    // Right IR LED & Receiver
  pinMode(8, OUTPUT); pinMode(7, OUTPUT);    // LED indicators -> output

  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12
}

void loop()                             // Main loop auto-repeats
{
  // Sniff

  int irLeft = digitalRead(10);         // Check for IR on left
  int irRight = digitalRead(3);         // Check for IR on right
```

```
  if((irLeft == 0) || (irRight == 0))
  {
    for(int i = 0; i < 5; i++)          // Repeat 5 times
    {
      digitalWrite(7, HIGH);            // Turn indicator LEDs on
      digitalWrite(8, HIGH);
      tone(4, 4000, 10);                // Sound alarm tone
      delay(20);                        // 10 ms tone, 10 between tones
      digitalWrite(7, LOW);             // Turn indicator LEDs off
      digitalWrite(8, LOW);
    }
  }

  // Roam

  irLeft = irDetect(9, 10, 38000);      // Check for object on left
  irRight = irDetect(2, 3, 38000);      // Check for object on right

  if((irLeft == 0) && (irRight == 0))   // If both sides detect
  {
    maneuver(-200, -200, 20);           // Backward 20 milliseconds
  }
  else if(irLeft == 0)                  // If only left side detects
  {
    maneuver(200, -200, 20);            // Right for 20 ms
  }
    else if(irRight == 0)               // If only right side detects
  {
    maneuver(-200, 200, 20);            // Left for 20 ms
  }
  else                                  // Otherwise, no IR detects
  {
    maneuver(200, 200, 20);             // Forward 20 ms
  }
}
```

Chapter 8. Robot Control with Distance Detection

In the previous chapter, we used the infrared LEDs and receivers to detect whether an object is in the BOE Shield-Bot's way without actually touching it. Wouldn't it be nice to also get some distance information from the IR sensors? There is a way to accomplish some rough, close-range detection with the very same IR circuit from the previous chapter. It's just enough information to allow the BOE Shield-Bot to follow a moving object without colliding into it.

The way is called *frequency sweep*. Frequency Sweep is the technique of testing a circuit's output using a variety of input frequencies.

Determining Distance with the IR LED/Receiver Circuit

This chapter uses the same IR LED/receiver circuits from the previous chapter.

You will also need:

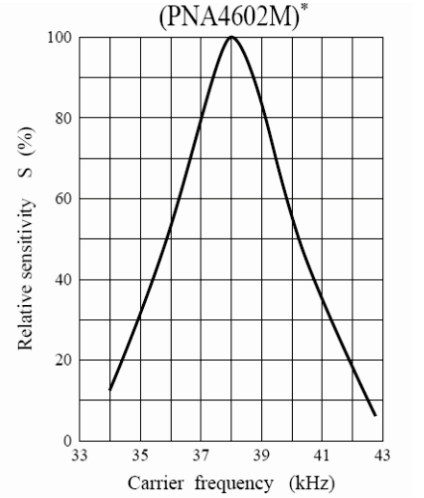
(1) ruler
(1) sheet of paper

- ✓ If the circuit from the last chapter is still built on your BOE Shield-Bot, make sure your IR LEDs are using the 2 kΩ resistors.
- ✓ If you already disassembled the circuit from the previous chapter, repeat the steps in Chapter 7, Activity 1 on page 218.
- ✓ Also, make sure to use TestBothIrAndIndicators to verify that the system is working properly.
- ✓ If you are working in different lightging condistions than you were before, use the IR Interference Sniffer to check for problems.

Activity 1: Testing the Frequency Sweep

Here's a graph from one specific brand of IR detector's datasheet (Panasonic PNA4602M; a different brand may have been used in your kit).

The graph shows that the IR detector is most sensitive at 38 kHz—its peak sensitivity —at the top of the curve. Notice how quickly the curve drops on both sides of the peak. This IR detector is much less sensitive to IR signals that flash on/off at frequencies other than 38 kHz. It's only half as sensitive to an IR LED flashing at 40 kHz as it would be to 38 kHz signals. For an IR LED flashing at 42 kHz, the detector is only 20% as sensitive. The further from 38 kHz an IR LED's signal rate is, the closer the IR receiver has to be to an object to see that IR signal's reflection.

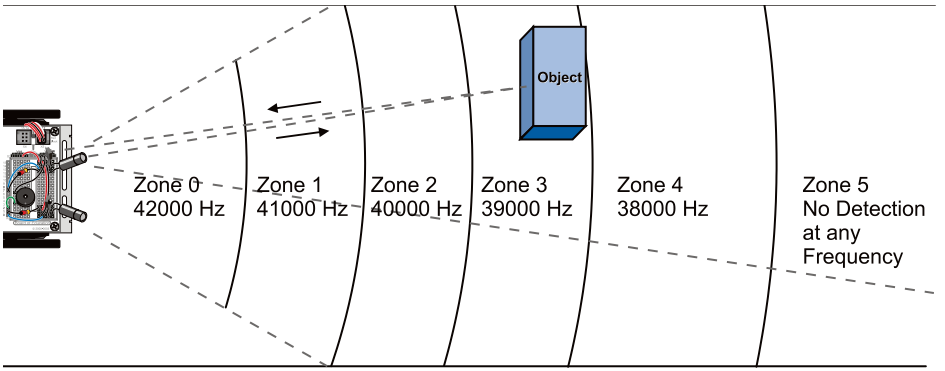


The most sensitive frequency (38 kHz) will detect the objects that are the farthest away, while less-sensitive frequencies can only detect closer objects. This makes rough distance detection rather simple: pick some frequencies, then test them from most sensitive to least sensitive. Try the most sensitive frequency first. If an object is detected, check and see if the next-most sensitive frequency detects it. Depending on which frequency makes the reflected infrared no longer visible to the IR detector, your sketch can infer a rough distance.

Programming Frequency Sweep for Distance Detection

The next diagram shows an example of how the BOE Shield-Bot can test for distance using frequency. Note this diagram is not to scale: the detection range only covers a short distance, and begins a few centimeters away from the robot.

In this example, an object is in Zone 3. That means that the object can be detected when 38000 and 39000 Hz is transmitted, but it cannot be detected with 40000, 41000, or 42000 Hz. If you were to move the object into Zone 2, then the object would be detected when 38000, 39000, and 40000 Hz are transmitted, but not with 41000 or 42000 Hz.Z



The **irDistance** function from the next sketch performs distance measurement using the technique shown above. Code in the main loop calls **irDistance** and passes it values for a pair of IR LED and receiver pins. For example, the **loop** function uses **int irLeft = irDistance(9, 10)** to check distance to an object in the left side of the BOE Shield-Bot's detection zone "field of vision."

```
// IR distance measurement function

int irDistance(int irLedPin, int irReceivePin)
{
  int distance = 0;
  for(long f = 38000; f <= 42000; f += 1000) {
    distance += irDetect(irLedPin, irReceivePin, f);
  }
  return distance;
}
```

The **irDetect** function returns a 1 if it doesn't see an object, or a zero if it does. The expression **distance += irDetect(irLedPin, irReceivePin, f)** counts the number of times the object is not detected. After the **for** loop is done, that **distance** variable stores the number of times the IR detector did not see an object. The more times it doesn't see an object, the further away it must be. So, the **distance** value the function returns represents the zone number in the drawing above.

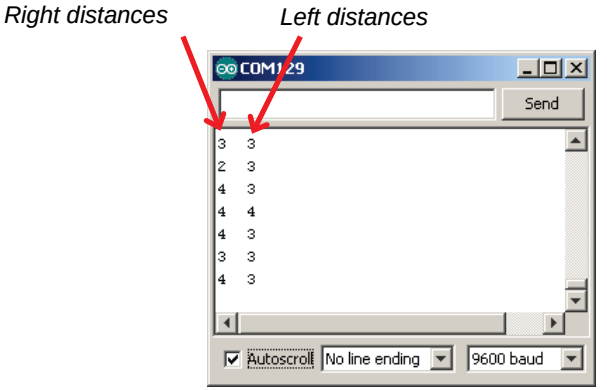
Displaying Both Distances

It's important to test that the detection distances are roughly the same for both IR LED/receiver pairs. They don't have to be identical, but if they are too far apart, the BOE Shield-Bot might have difficulty following an object in front of it because it will keep trying to turn to one side. More subsystem testing!

A common cause of mismatched distance measurement is mismatched resistors used with the IR LEDs. For example, if one side's IR LED has a 1 kΩ resistor and the other has a 2 kΩ resistor, one side will need objects to be much closer to see them. Another possibility, though rare, is that one IR detector is far more sensitive than the other. In that case, a larger resistor can be used in series with the IR LED on that side to make its IR headlight dimmer and correct the mismatched measurements.

Example Sketch – DisplayBothDistances

This screen capture shows some detection zone measurements from DisplayBothDistances in the Serial Monitor. Though there's fluctuation in the values, commonly called *noise*, what matters is that the numbers match, or are off by only 1, in each pair of measurements.



- ✓ Create, save, and run the sketch DisplayBothDistances on your Arduino.
- ✓ Use a box, book, bottle, or similar object as a target for the left distance detector.
- ✓ Start by moving the object toward and away from the BOE Shield-Bot until you find the small range where a small movement will result in a change in distance measurement.
- ✓ Find and record the midpoint of each distance detection zone.
- ✓ Repeat for the right IR detector.

If it turns out that the detection range of one side is twice as far as the other (or more), check the resistors connected to the IR LEDs. You may have a mismatch there; make sure both resistors are 2 kΩ (red-black-red). If there isn't a mismatch, try adjusting IR LED resistor values until the detection ranges of both sides are in the same neighborhood.

```
/*
 * Robotics with the BOE Shield - DisplayBothDistances
 * Display left and right IR states in Serial Monitor.
 * Distance range is from 0 to 5.  Only a small range of several cm
 * in front of each detector is measureable.  Most of it will be 0 (too
 * close) or 5 (too far).
 */

void setup()                            // Built-in initialization block
{
  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  pinMode(10, INPUT);  pinMode(9, OUTPUT);   // Left IR LED & Receiver
  pinMode(3, INPUT);  pinMode(2, OUTPUT);    // Right IR LED & Receiver

  Serial.begin(9600);                   // Set data rate to 9600 bps
}

void loop()                             // Main loop auto-repeats
{
  int irLeft = irDistance(9, 10);       // Measure left distance
  int irRight = irDistance(2, 3);       // Measure right distance

  Serial.print(irLeft);                 // Display left distance
  Serial.print("  ");                   // Display spaces
  Serial.println(irRight);              // Display right distance

  delay(100);                           // 0.1 second delay
}

// IR distance measurement function

int irDistance(int irLedPin, int irReceivePin)
{
  int distance = 0;
  for(long f = 38000; f <= 42000; f += 1000) {
    distance += irDetect(irLedPin, irReceivePin, f);
  }
  return distance;
}

// IR Detection function

int irDetect(int irLedPin, int irReceiverPin, long frequency)
{
  tone(irLedPin, frequency, 8);         // IRLED 38 kHz for at least 1 ms
  delay(1);                             // Wait 1 ms
```

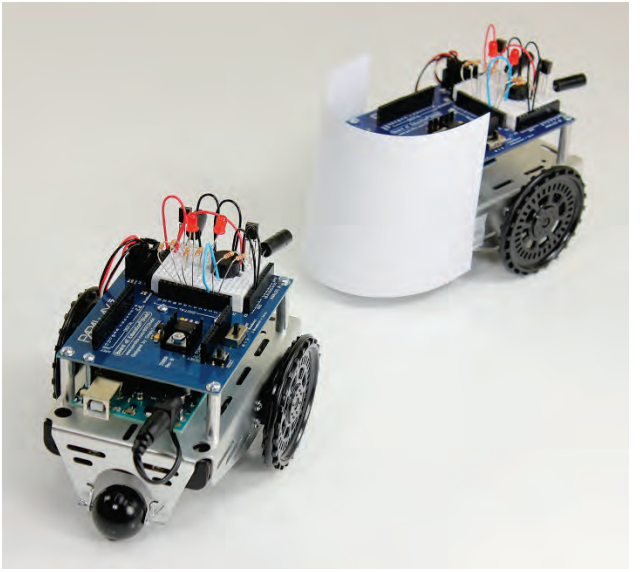
```
  int ir = digitalRead(irReceiverPin);  // IR receiver -> ir variable
  delay(1);                             // Down time before recheck
  return ir;                            // Return 1 no detect, 0 detect
}
```

Your Turn – More Distance Tests

- ✓ Try measuring the detection range for objects with different colors and textures. Which colors and surfaces are easiest to detect? Which are most difficult?

Activity 2: BOE Shield-Bot Shadow Vehicle

For a BOE Shield-Bot to follow a leader-object, it has to know the rough distance to the leader. If the leader is too far away, the sketch has to be able to detect that and move the BOE Shield-Bot forward. Likewise, if the leader is too close, the sketch has to detect that and move the BOE Shield-Bot backward. The purpose of the sketch is to make the BOE Shield-Bot maintain a certain distance between itself and the leader-object.



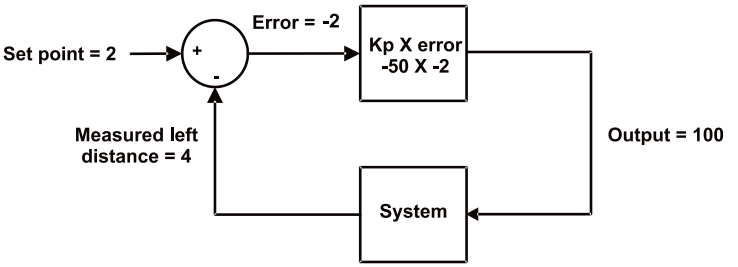
Some Control System Vocabulary

When a machine is designed to automatically maintain a measured value, it generally involves a *control system*. The value that the system is trying to maintain is called the *set point*. Electronic control systems often use a *processor* to take sensor measurements and respond by triggering mechanical *actuators* to return the machine to the set point.

Our machine is the BOE Shield-Bot. The measured value we want it to maintain is the distance to the leader-object, with a set point of 2 (for zone 2). The machine's processor is the Arduino. The IR LED/receiver pairs are the sensors that take distance value measurements to the leader-object. If the measured distance is different from the set-point distance, the servos are the mechanical actuators that rotate to move the BOE Shield-Bot forward or backward as needed.

A Look Inside Proportional Control

Closed loop control—repeatedly measuring a value and adjusting output in proportion to error for maintaining a set point—works very well for the BOE Shield-Bot shadow vehicle. In fact, the majority of the control loop shown in the diagram below reduces to just one line of code in a sketch. This block diagram describes the proportional control process that the BOE Shield-Bot will use to control the wheel speed based on detection distance measured with the IR LED/receiver pairs.

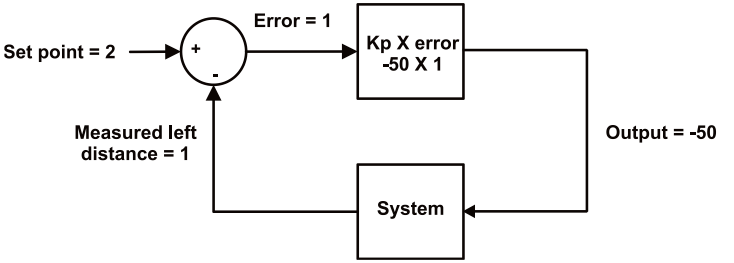


This block diagram could apply to either the left IR distance sensor and servo output or the right. In fact, your code will maintain two identical control loops, one for each side of the BOE Shield-Bot. Let's walk through this example.

In the upper left, the set point = 2; we want the BOE Shield-Bot to maintain a zone-2 distance between itself and its leader-object. Below that, the measured distance is zone 4, so the leader-object is too far away. The arrows towards the symbols in the circle (called a *summing junction*) indicate that you add (+) the set point and subtract (-) the measured distance together to get the *error*, in this case 2 – 4 = -2.

Next, the error value feeds into the top square—an *operator block*. This block shows that the error gets multiplied by -50, a *proportionality constant* (Kp). In this example, the operator block gives us -2 × -50 = 100, so 100 is the output. In a sketch, this output value gets passed to the **`maneuver`** function. It turns the servo full speed forward to move the BOE Shield bot closer to the leader-object.

The next block diagram shows another example. This time, the measured distance is 1, meaning the leader-object is too close. So, the error is 1, and 1×–50 = -50. Passing -50 to the maneuver function turns the servo half-speed in reverse, backing the BOE Shield-Bot away from the leader-object.



The next time through the loop, the measured distance might change, but that's okay. Regardless of the measured distance, this control loop will calculate a value that will cause the servo to move to correct any error. The correction is always *proportional* to the error. The two calculations involved:

set point – measured distance = error; error x Kp = output for maneuver

…can be easily combined and re-ordered into one expression for your sketch:

Output for maneuver = (Distance set point – Measured distance) x Kp

If you want to take a look at the sketch in detail, see How FollowingShieldBot Works on page 258.

Your Turn – Verify Control Loop with Other Distances

To prove that the proportional control loop responds correctly to all six measured distances, fill in the table below. The block diagrams have been solved for two of the six conditions, so you've only got four to try.

Condition	Measured Distance	Set Point	Error (set point–measured)	Output (Kp × error)	Maneuver Result
	0	2			
Too close	1	2	2 – 1 = 1	2 × –50 = -50	Slow reverse
Just right	2	2			
	3	2			
Way too far	4	2	2 – 4 = -2	-2 × -50= -100	Fast forward
	5	2			

Example Sketch: FollowingShieldBot

The FollowingShieldBot sketch repeats the proportional control loops just discussed at a rate of about 25 times per second. So, all the proportional control calculations and servo speed updates happen 25 times each second. The result is a BOE Shield-Bot that will follow your hand, a book, or another robot.

- ✓ Create, save, and run the sketch FollowingShieldBot.
- ✓ Point the BOE Shield-Bot at an 8 ½ x 11" sheet of paper held in front of it as though it's a wall-obstacle. The BOE Shield-Bot should maintain a fixed distance between itself and the sheet of paper. (It will dance around a little because of the sensor noise mentioned earlier.)
- ✓ Rotate the sheet of paper slightly; the BOE Shield-Bot should rotate with it.
- ✓ Try using the sheet of paper to lead the BOE Shield-Bot around. The BOE Shield-Bot should follow it.
- ✓ Move the sheet of paper too close to the BOE Shield-Bot, and it should back up, away from the paper.

```
/*
 * Robotics with the BOE Shield - FollowingShieldBot
 * Use proportional control to maintain a fixed distance between
 * BOE Shield-Bot and object in front of it.
 */

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

const int setpoint = 2;                 // Target distances
const int kpl = -50;                    // Proportional control constants
const int kpr = -50;

void setup()                            // Built-in initialization block
{
  pinMode(10, INPUT);  pinMode(9, OUTPUT);   // Left IR LED & Receiver
  pinMode(3, INPUT);  pinMode(2, OUTPUT);    // Right IR LED & Receiver

  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12
}

void loop()                             // Main loop auto-repeats
{
  int irLeft = irDistance(9, 10);       // Measure left distance
  int irRight = irDistance(2, 3);       // Measure right distance
```

```
  // Left and right proportional control calculations
  int driveLeft = (setpoint - irLeft) * kpl;
  int driveRight = (setpoint - irRight) * kpr;

  maneuver(driveLeft, driveRight, 20);  // Drive levels set speeds
}

// IR distance measurement function

int irDistance(int irLedPin, int irReceivePin)
{
  int distance = 0;
  for(long f = 38000; f <= 42000; f += 1000) {
    distance += irDetect(irLedPin, irReceivePin, f);
  }
  return distance;
}

// IR Detection function

int irDetect(int irLedPin, int irReceiverPin, long frequency)
{
  tone(irLedPin, frequency, 8);         // IRLED 38 kHz for at least 1 ms
  delay(1);                             // Wait 1 ms
  int ir = digitalRead(irReceiverPin);  // IR receiver -> ir variable
  delay(1);                             // Down time before recheck
  return ir;                            // Return 1 no detect, 0 detect
}

void maneuver(int speedLeft, int speedRight, int msTime)
{
  // speedLeft, speedRight ranges: Backward  Linear  Stop  Linear   Forward
  //                               -200      -100......0......100       200
  servoLeft.writeMicroseconds(1500 + speedLeft);   // Left servo speed
  servoRight.writeMicroseconds(1500 - speedRight); // Right servo speed
  if(msTime==-1)                                   // If msTime = -1
  {
    servoLeft.detach();                            // Stop servo signals
    servoRight.detach();
  }
  delay(msTime);                                   // Delay for msTime
}
```

How FollowingShieldBot Works

FollowingShieldBot declares three global constants: **`setpoint`**, **`kpl`**, and **`kpr`**. Everywhere you see **`setpoint`**, it's actually the number 2 (a constant). Likewise, everywhere you see **`kpl`**, it's actually the number -50. Likewise with **`kpr`**.

```
const int setpoint = 2;                 // Target distances
const int kpl = -50;                    // Proportional control constants
const int kpr = -50;
```

The convenient thing about declaring constants for these values is that you can change them in one place, at the beginning of the sketch. The changes you make at the beginning of the sketch will be reflected everywhere these constants are used. For example, by changing the declaration for **kpl** from -50 to -45, every instance of **kpl** in the entire sketch changes from -50 to -45. This is exceedingly useful for experimenting with and tuning the right and left proportional control loops.

The first thing the **loop** function does is call the **irDistance** function for current distance measurements and copies the results to the **irLeft** and **irRight** variables.

```
void loop()                                     // Main loop auto-repeats
{
  int irLeft = irDistance(9, 10);               // Measure left distance
  int irRight = irDistance(2, 3);               // Measure right distance
```

Remember the simple control loop calculation?

Output for maneuver = (Distance set point – Measured distance) x Kp

The next two lines of code perform those calculations for the right and left control loops, and store the output-for-maneuver results to variables named **driveLeft** and **driveRight**.

```
  // Left and right proportional control calculations
  int driveLeft = (setpoint - irLeft) * kpl;
  int driveRight = (setpoint - irRight) * kpr;
```

Now, **driveLeft** and **driveRight** are ready to be passed to the **maneuver** function to set the servo speeds.

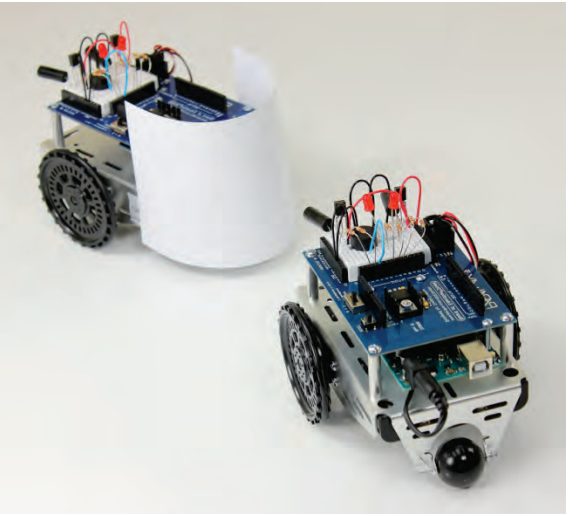
```
  maneuver(driveLeft, driveRight, 20);        // Drive levels set speeds
}
```

Since each call to maneuver lasts for 20 ms, it delays the **loop** function from repeating for 20 ms. The IR distance detection takes another 20 ms, so the **loop** repetition time is about 40 ms. In terms of sampling rate, that translates to 25 samples per second.

Sampling Rate vs. Sample Interval: The *sample interval* is the time between one sample and the next. The *sampling rate* is the frequency at which the samples are taken. If you know one term, you can always figure out the other: sampling rate = 1 ÷ sample interval.

Follow the Leader

Here's a leader BOE Shield-Bot followed by a shadow BOE Shield-Bot. The lead robot is running a modified version of FastIrRoaming, with maneuver speeds reduced to +/- 40. The shadow BOE Shield-Bot is running FollowingShieldBot. One lead robot can string along a chain of 6 or 7 shadow robots. Just add the paper panel to the rest of the shadow BOE Shield-Bots in the chain.



- ✓ If you are working on your own with one BOE Shield-Bot, you will be the leader! The leader-object can be a book, bottle or even just your hand.
- ✓ If you are part of a class with two or more BOE Shield-Bots, mount a paper panel around the tail and both sides of a lead robot to make it more visible to the shadow robots, like in the picture. If you are making a chain of shadow robots, put a paper panel on each of them too.
- ✓ Program the lead BOE Shield-Bot with SlowerIrRoamingForLeaderBot.
- ✓ Program each shadow BOE Shield-Bot with FollowingShieldBot. Each shadow robot's IR LEDs should be pointing slightly to the left and right, and level with horizontal (not up or down).
- ✓ Place a shadow BOE Shield-Bot behind the lead BOE Shield-Bot or other leader-object. The shadow BOE Shield-Bot should follow the leader at a fixed distance, so long as it is not distracted by another object such as a hand or a nearby wall.

```
/*
 * Robotics with the BOE Shield - SlowerIrRoamingForLeaderBot
 * Adaptation of RoamingWithWhiskers with IR object detection instead of
 * contact switches
 */

#include <Servo.h>                      // Include servo library

Servo servoLeft;                        // Declare left and right servos
Servo servoRight;

void setup()                            // Built-in initialization block
{
  pinMode(10, INPUT);
  pinMode(9, OUTPUT);                   // Left IR LED & Receiver
  pinMode(3, INPUT);
  pinMode(2, OUTPUT);                   // Right IR LED & Receiver

  tone(4, 3000, 1000);                  // Play tone for 1 second
  delay(1000);                          // Delay to finish tone

  servoLeft.attach(13);                 // Attach left signal to pin 13
  servoRight.attach(12);                // Attach right signal to pin 12
}

void loop()                             // Main loop auto-repeats
{

  int irLeft = irDetect(9, 10, 38000);  // Check for object on left
  int irRight = irDetect(2, 3, 38000);  // Check for object on right

  if((irLeft == 0) && (irRight == 0))   // If both sides detect
  {
    maneuver(-40, -40, 20);             // Backward 20 milliseconds
  }

  else if(irLeft == 0)                  // If only left side detects
  {
    maneuver(40, -40, 20);              // Right for 20 ms
  }
  else if(irRight == 0)                 // If only right side detects
  {
    maneuver(-40, 40, 20);              // Left for 20 ms
  }
  else                                  // Otherwise, no IR detects
  {
    maneuver(40, 40, 20);               // Forward 20 ms
  }
}

int irDetect(int irLedPin, int irReceiverPin, long frequency)
{
  tone(irLedPin, frequency, 8);         // IRLED 38 kHz for at least 1 ms
  delay(1);                             // Wait 1 ms
```

```
  int ir = digitalRead(irReceiverPin);  // IR receiver -> ir variable
  delay(1);                             // Down time before recheck
  return ir;                            // Return 1 no detect, 0 detect
}

void maneuver(int speedLeft, int speedRight, int msTime)
{
  // speedLeft, speedRight ranges: Backward  Linear  Stop  Linear   Forward
  //                               -200      -100......0......100       200
  servoLeft.writeMicroseconds(1500 + speedLeft);   // Left servo speed
  servoRight.writeMicroseconds(1500 - speedRight); // Right servo speed
  if(msTime==-1)                                   // If msTime = -1
  {
    servoLeft.detach();                            // Stop servo signals
    servoRight.detach();
  }
  delay(msTime);                                   // Delay for msTime
}
```

Your Turn – Experiment with the Constants

You can adjust the setpoint and proportionality constants to change the shadow BOE Shield-Bot's behavior. Use your hand or a piece of paper to lead the shadow BOE Shield-Bot while doing these exercises:

- ✓ Try running FollowingShieldBot using other values of **`kpr`** and **`kpl`** constants, ranging from 15 to 100. Note the difference in how responsive the BOE Shield-Bot is when following an object.
- ✓ Try making adjustments to the value of the **`setpoint`** constant. Try values from 0 to 4.

You might notice some odd behaviors. For example, if the setpoint is 0, it won't back up. Want to figure out why?

- ✓ Repeat the control loop exercises from Activity #1 with the setpoint at zero. Can any measured distance cause it to back up with a set point of zero?

Activity 3: What's Next?

Congratulations! You've made it to the end of the book. Now that you've mastered the basics, you're ready to start exploring and adding capabilities with your BOE Shield-Bot! Links to these ideas and more are available from the Arduino-Friendly Products page at www.parallax.com/product/arduino.

A few BOE Shield-Bot projects are shown on the next two pages.

Shield-Bot Roaming with PING)))

A PING))) Ultrasonic Distance Sensor on a servo turret mounts on the front of the Shield-Bot, for autonomous navigation.

The ultrasonic distance sensor detects objects up to 10 feet (3 m) away. When an obstacle is detected, the Shield-Bot stops, sweeps the sensor from side to side, and then turns down the clear path.

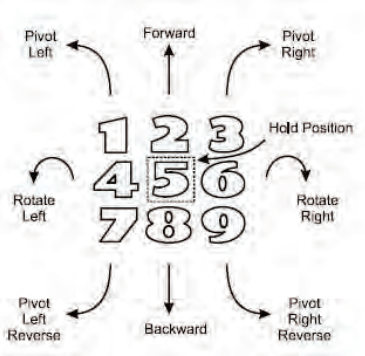
Ultrasonic distance sensors work well in strong sunlight where infrared sensors are overwhelmed, making this a good choice for bright environments or direct sunlight.

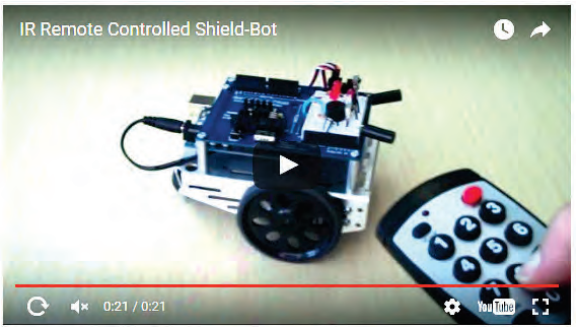
http://learn.parallax.com/tutorials/projects/shield-bot-roaming-ping

IR Remote Controlled Shield-Bot

A simple universal remote control, programmed to use the Sony protocol, can also control your Shield-Bot using the same IR Receiver included in the kit. The project code allows you to switch between 3 modes on the go: remote control, autonomous roaming with object avoidance, and follow-the-leader behavior.

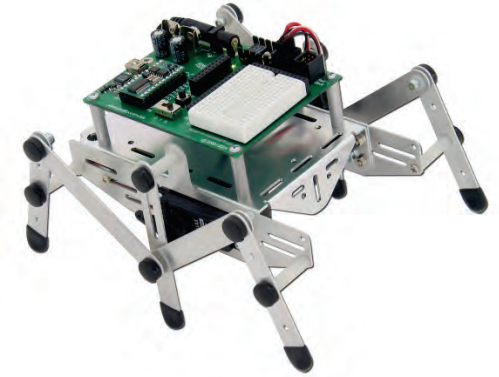
http://learn.parallax.com/tutorials/projects/ir-remote-controlled-shield-bot

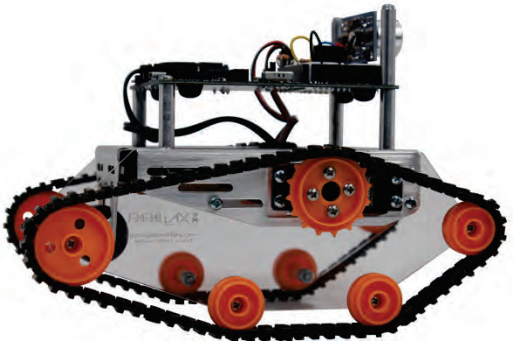




Hardware Add-Ons

The Crawler Legs (left, #30055) and Tank Treads (right, #28106) give alternate modes of locomotion to your Shield-Bot robot. You may run the same navigation code, but the ground speed will be different so you may need to adjust the duration of different maneuvers.





Chapter 8 Summary

This chapter used the infrared IR LED/receiver pairs for simple distance detection, to make a BOE Shield-Bot shadow vehicle. Now-familiar skills combined with some new concepts got the job done:

Electronics

- Looking at the IR receiver's peak sensitivity
- Exploiting the IR receiver's sensitivity curve properties with a frequency sweep to detect distance
- Changing series resistor values to match up the sensitivity of two IR LED receiver pairs—making an adjustment in hardware

Programming

- Writing a routine with an indexing **`for`** loop to generate a frequency sweep
- Writing a sketch that uses proportional control loops
- Using **`const int`** to set up a proportional control loop's set point and proportionality constants
- Changing a control loop's proportionality constants to fine-tune the control system—making an adjustment in software
- What sampling rate and sampling interval are, and how they relate to each other

Robotics Skills

- Setting up a closed-loop proportional control system with a processor, sensors and actuators for autonomous shadow vehicle navigation

Engineering Skills

- Reading a block diagram for a simple closed loop proportional control system, with a summing junction and operator blocks
- Understanding and using a closed loop control system's set point, error, proportionality constant, and output value
- More subsystem testing and troubleshooting

Chapter 8 Challenges

Questions

1. What would the relative sensitivity of the IR detector be if you use **`tone`** to send a 35 kHz signal? What is the relative sensitivity with a 36 kHz signal?
2. What keyword is used to declare a constant?

3. What statement is used to sweep through the list of frequencies used for IR distance measurements?
4. What's a summing junction?
5. In our closed loop proportional control block diagrams, what two values are used to calculate the error term?
6. How do delays in the **loop** function contribute to setting the sampling rate?

Exercise

1. Write a segment of code that does the frequency sweep for just four frequencies instead of five.

Project

1. Write a sketch that allows you to hold your hand in front of the BOE Shield-Bot and push it backwards and forwards with no turning.

Question Solutions

1. The relative sensitivity at 35 kHz is 30%. For 36 kHz, it's 50%.
2. Precede a variable declaration with the **const** keyword.
3. A **for** loop that starts indexing at 38000 and increases by 1000 with each repetition.
4. A summing junction is a part of a block diagram that indicates two inputs are added together (or one subtracted from another) resulting in an output.
5. The error term is the measured level subtracted from the desired set point level.
6. If a distance sample is taken with each repetition of the **loop** function, then the delays more or less determine how long it takes between each sample. That's called the sample interval, and 1 ÷ sample interval = sampling rate.

Exercise Solution

1. Just reduce the **for** statement's condition from **f <=42000** to **f <= 41000.**

```
  for(long f = 38000; f <= 42000; f += 1000) {
    distance += irDetect(irLedPin, irReceivePin, f);
  }
// Declarations
const int setpoint = 2;    // Target distances
const int kpl = -45;       // Proportional control constants
const int kpr = -55;
```

Project Solution

1. One quick and simple solution would be to average the **`driveLeft`** and **`driveRight`** values in the FollowingShieldBot sketch. The resulting single value can be applied both left and right speed parameters in the maneuver call.

```
void loop()                           // Main loop auto-repeats
{
  int irLeft = irDistance(9, 10);     // Measure left distance
  int irRight = irDistance(2, 3);     // Measure right distance

  // Left and right pr67oportional control calculations
  int driveLeft = (setpoint - irLeft) * kpl;
  int driveRight = (setpoint - irRight) * kpr;

  int drive = (driveLeft + driveRight)/2; // Average drive levels

  maneuver(drive, drive, 20);         // Apply same drive to both

  delay(10);                          // 0.1 second delay
}
```

Index

For your convenience, this book is available as a free, searchable PDF download from the #122-32335 product page at www.parallax.com.

-- decrement operator 34
! = not equal operator 165
! NOT operato XE "operators:/ division" r 27
#include 57
% modulus operator 27
&& boolean AND operator 31
* multiplication 27
/ division 27
/* and */ 36
// line comment 36
|| boolean OR operator 31
+ addition 27
++ increment operator 34, 35
++ post increment 34
< less than operator 31
== compare-equals operator 31
> greater than operator 30
38 kHz 250
3-position switch 61
active-low output 152
actuators 255
adding a library 57
addition 27
ambient light 179
amperes 186
amps 52
ANALOG IN sockets 50
analog sensor 184
analog to digital conversion 183
analogRead function 183
anode 48, 51, 220
Arduino IDE software 16
Arduino language
  analogRead function 183
  byte 26
  chaining else if 32
commenting code 36
constrain function 204
DEC 26
delay function 22
digitalRead 151
digitalWrite 53
disableServos function 124
example function 119
float 24
for loop 32, 33
go function 138
HIGH 151
if...else 30
include 57
infinite loop 35
INPUT 52
int24
irDetect function 222
irDistance function 251
long 27
loop 20
LOW 151
maneuver function 125
map function 185
operators ..... *See* operators
OUTPUT 52
PI 29
pinMode 52
pinMode function 151
pitch function 121
println 23
rcTime function 194
Serial library 20
Serial.println() 35
Serial.read 96
Servo library 57
setup 20, 21
sizeof function 131
sqrt 24

switch/case....138
tone function....221
true....35
variable type....24
void setup....21
volts function....183
while loop....32, 34, 35
writeMicroseconds....57
Arduino language 1.0....10
Arduino Language Reference....24
Arduino Uno....9, 10
Arduino, mounting on BOE Shield....44
area of a circle....29
array....128
arrays and navigation....133
arrays, character....136
artificial intelligence....162
ASCII code....25
assesment materials....8
assignment operator....27
Atmel microcontroller....50
attach a servo....57
autonomous robot navigation....157
backwards driving....107
ballast, light fixture....224
BASIC Stamp....7
batteries....12, 63
battery pack....62
installation....81
baud rate....20
in Serial Monitor....18
baud rate in Serial Monitor....19
bill of materials....11
binary sensor....184
blink rate....54
block....22
block comment....36
block diagram, control system....255, 256
Board of Education Shield....9
mounting to chassis....84
Power switch....51
prototyping area....50
set up....42
BOE Shield-Bot
assembly....76
orientation....85, 103
Boe-Bot....7
Boe-Bot, retrofit to Arduino....9
boolean operators....31
brain....9
breadboard....49
brownout....89
buffer, serial....96
building circuits....51
bumper switches....*See* whiskers
byte....28
byte variable type....26
calling functions....119
capacitance measurements....189
capacitor....189
Capek, Karel....6
capitalization....18, 19
case sensitivity....18, 19
cathode....48, 51, 220
centering servos....63
chain else if statements....32
char variable type....24
character arrays....136
charge transfer circuit....191
chassis parts list....12
circle....29
circuit diagrams
LEDs....51
LEDs with IR object detection....225
photoresistor charge transfer....190
phototransistor voltage....*See*
piezospeaker....90
prototyping area....50
servos....62
whisker LED test....154
whisker switches....149
circumference of a circle....29
closed loop control....255
code block....22

Codebender 10, 16
collector, phototransistor 177
comment 21
  block 36
  line 36
  purpose of 36
comparison of equals 31
compiler 21, 25
compound arithmetic operators 34
computer buffer overruns 154
connected in parallel 192
connected in series 186
const declarations 37
constants 29, 37
constrain function 204
continuous rotation servo
  and writeMicroseconds 69
  center calibration 63
  chassis mounting options 80
  connecting to BOE Shield 61
  parts of 60
  PWM control 66
  run time 70
  Servo library 57
  speed transfer curve 93
  testing after assembly 86
  troubleshooting 88
control system 254
Crawler Legs 264
curly braces 18, 19, 21, 22, 119, 135
current 47, 52
current in a circuit 185
current, in Ohm's Law 186
data types 27
dead reckoning 112
DEC 26
decimal points 29
decimal values 24
decisions 30
delay function 22
detach a servo 57
digital cameras 217, 218
DIGITAL sockets 50
digital value 184
digitalRead 151
digitalWrite 53
diode 48
disableServos function 124
distance detection zone graphic 251
distance traveled calculations 112
division 27, 28
documenting code 36
drop-off detector 238
Duemilanove 10
educator resources 8
electric current 52
electrical tape 239
electronic components list 14
element, array 128
emitter, phototransistor 177
encoders 116
error, control system 255
escaping corners 162
example function 119
EXT pin 10
farad 189
float 28
float variable type 24
floating-point 29
fluorescent lights 217
follow the leader 260
for loop 32
for loop structure 33
forward slashes // 21
frequencies of music notes 129
frequency 89, 91
frequency sweep 249
function
  call 119
  call 20
  call with parameter passing 120
  definition 21
  meaning 20

nesting calls 154
General Motors 6
global variables 26
GND 52
GND sockets 50
go function 138
graph, voltage decay 196
graphic light display 200
graphic, distance detection zone 251
greater than operator 30
ground 52
halogen lamps 178
Hardware Add-ons
Crawler Legs 264
PING Ultrasonic Distance Sensor on a servo turret 263
remote control 264
hardware adjustment, servo 109
hardware parts list 13
hertz 91
HIGH signal 151
human hearing range 91
Hz 91
if…else 30
incandescent lamps 178
include 57
infinite loop 35
infrared 176
infrared interference 217
infrared LEDs 219
infrared object detector circuits 220
infrared receivers 218
initialization of for loops 33
input pin 151
input pin mode 52
installing software 16
instance of an object 105
int variable type 24, 28
integer values 24
interference, infrared 228
inventory of parts 11
IR navigation 232
irDetect function 222
irDistance function 251
iterative process 111
Karel Capek 6
kilo-ohm 47
kit options 9
kit parts list 11
Kp, proportionality constant 255
leads 47
LED circuit 51
LED circuits for IR object detectors 225
LED light shield 219
LED light-emitting diode 48
less than operator 31
library, adding to a project 57
light spectrum 178, 216
line comment 36
local variables 26, 121
in for loops 33
long 28
long variable type 27
LOW signal 151
maneuver function 125
maneuvers in functions 123
map function 185
measurements, noise in 252
measurements, normalized differential . 197
measuring distance vs speed 113
Mega 10, 44
microfarad 189
microsecond 58
millisecond 23, 58
modulation 69
modulus 27, 28
multiple conditions 31
multiplication 27, 28
music note frequencies 129
nanometers 178
navigate by touch 157
navigation with arrays 133

navigation with distance detection..... 254
navigation, following object..... 254
nearsighted object detection..... 239
negative numbers 28
nesting function calls 154
noise, in measurements..... 252
normalized differential measurement.... 197
normally open switch..... 151
Not (!) Operator 227
note frequencies 129
not-equal operator 165
object avoidance..... 232
object detection range..... 230
object instance..... 105
ohm 47
Ohm's Law 186
operator block, control system..... 255
operators 27
-- decrement 34
! NOT 227
!= not equal..... 165
% modulus..... 27
&& boolean AND..... 31
/ division..... 27
|| boolean OR 31
+ addition 27
++ increment..... 34
++ post increment 34
++ pre-increment 35
< less than 31
= assign-equals..... 31
== compare-equals..... 31
> greater than 31
compound arithmetic..... 34
greater than 30
post increment..... 138
-subtraction..... 27
orientation of BOE Shield-Bot..... 103
oscilloscope..... 196
output pin mode 52
parallel, connected in 192
parameter 20
parentheses 18, 19, 20, 33
parentheses, empty 21
part numbers..... 11
passing parameters in function call 120
PBASIC 7
peak sensitivity, IR receiver..... 250
phototransistor 177
phototransistor charge transfer circuit... 190
phototransistor voltage circuit..... 179
PI constant..... 29
piano key frequencies 129
piezoelectric element..... 89
piezospeaker 89
piezospeaker circuit 90
PING Ultrasonic Distance Sensor..... 263
pinMode function..... 52, 151
pitch function 121
pivoting 108
post increment operator..... 138
poster board..... 239
potentiometer..... 64
power supply..... 62
power switch..... 51, 61
power terminals..... 50
println..... 23
processor, in control system 255
programs ..... *See* sketch, sketches
proportional control..... 255
proportionality constant 255
prototyping area 50
pulse train 66
pulse width modulation 69
PWM..... 69
PWR_SEL 10
QT Circuit..... 192
ramping 116
rcTime function 194
rcTime voltage decay graph 196
remote control 264
remote controller..... 228
reset indicator circuit 90
resistance, in Ohm's Law..... 186

resistor 47
color code value 48
tolerance 47
resolution, A/D 183
result values 28
Retrofit Kit 9, 43
Robot Hardware Refresher Packs 13
robot kit parts pictures 11
robot parts list 11
Robot Shield Kit with Arduino 9
ROBOTC 84
Robotics Shield Kit for Arduino 9
Rossum's Universal Robots 6
RPM 66
sampling rate, object detection 235
sampling, interval vs rate 260
schematic symbol 47
capacitor 189
infrared receiver 218
phototransistor 177
piezospeaker 89
resistor 47
servo 62
switch, whisker 149
sensor, analog 184
sensor, binary vs analog 184
serial buffer 96
Serial Monitor baud rate 18, 19
series resistance 230
series, connected in 186
servo
hardware adjustment 109
ramping 116
software adjustment 109
transfer curve graph 93
servo circuits 62
Servo library 57
servo turret 263
set point, in control system 255
setup 21
shadow vehicle 254
signed numbers 28
SimplyTronics 7
single-pole, single throw switch 151
sizeof function 131
sketch examples
AvoidTableEdge 240
BothServosStayStill 59
Chapter 1, Project 1 40
Chapter 1, Project 2 41
Chapter 2, Project 1 74
Chapter 3, Project 1 101
Chapter 3, Project 2 102
Chapter 4, project 2 - Circle 144
Chapter 4, project 2 - Triangle 145
Chapter 5 Project 1 172
Chapter 5, Exercise 1 170
Chapter 5, project 2 – WhiskerCircle 174
Chapter 6, Project 1 212
Chapter 6, Project 3 213
Circumference 29
ControlWithCharacters 136
CountToTenDocumented 36
DisplayWhiskerStates 152
EscapingCorners 162
FastIrRoaming 236
FollowingShieldBot 257
ForwardLeftRightBackward 107
ForwardOneSecond 114
ForwardTenSeconds 110
ForwardThreeSeconds 104
FunctionCallWithParameter 122
HaltUnderBrightLight 182
Hello 17
HelloRepeated 23
HighLowLed 53
IrInterferenceSniffer 229
LeftLedOn 231
LeftServoClockwise 66
LeftServoStayStill 58
LightSeekingDisplay 205
LightSeekingShieldBot 207
LightSensorDisplay 200
LightSensorValues 199
ManeuverSequence 134
MovementsWithSimpleFunctions 123
PhototransistorVoltage 181

PlayAllNotesInArray 132
PlayNotesWithLoop..... 131
PlayOneNote 130
RightServoClockwise 67
RightServoStayStill 65
RoamingWithIr 233
RoamingWithWhiskers..... 158
ServoSlowMoCcw..... 56
ServosOppositeDirections 68
SimpleDecisions 30
SimpleFunctionCall..... 120
SimpleMath..... 28
SlowerIrRoamingForLeaderBot 261
StartAndStopWithRamping 117
StartResetIndicator 91
StoreRetrieveLocal 25
TestBothIrAndInciators 226
TestLeftIR 223
TestServoSpeed..... 95
TestWhiskersWithLeds..... 156
sketches 10, 16
software adjustment, servo..... 109
software options..... 10
Arduino IDE 16
solderless breadboard 49
sqrt..... 24
straight line driving 109
subsystem testing 65
subtraction..... 27
summing junction 255
sunlight 178
switch/case 138
table edge detector 238
timing diagram..... 54
tolerance, resistor..... 47
tone function 221
tone function duration..... 222
transfer curve, servo..... 93
transistor..... 177
true, predefined constant 35
tuning turn maneuvers.....111
TV remote.....217, 228
ultrasonic distance sensor.....263
Uno.....10
unsigned numbers.....28
variable type.....24
byte 26
choosing for return value 28
int24
long.....27
variables 24
array 128
declarations.....24
global vs local 26
types.....25
Vin 50
Vin sockets 50
void setup.....21
voltage.....52
voltage decay graph 196
voltage regulator.....51
voltage, in Ohm's Law 186
volts.....52
volts function 183
wheel rotation and direction 104
wheels, attaching 83
while loop.....32, 34, 35
whisker LED test circuits.....154
whisker sensor circuits 149
whiskers 147
writeMicroseconds.....57
us parameter range 111
zero indexed, array.....128
µF.....189
Ω omega or ohm 47