Robotics with the Boe-Bot

Student Guide

VERSION 3.0

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

ISBN 9781928982531

3.0.0-10.11.10-HKTP

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 BASIC Stamp 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

Preface

Robots are used in the auto, medical, and manufacturing industries, in all manner of exploration vehicles, and, of course, in many science fiction films. The word "robot" first appeared in a Czechoslovakian satirical play, Rossum's Universal Robots, by Karel Capek in 1920. Robots in this play tended to be human-like. From this point onward, it seemed that many science fiction stories involved these robots trying to fit into society and make sense out of human emotions. This changed when General Motors installed the first robots in its manufacturing plant in 1961. These automated machines presented an entirely different image from the “human form” robots of science fiction.

Building and programming a robot is a combination of mechanics, electronics, and problem solving. What you're about to learn while doing the activities and projects in this text will be relevant to real-world applications that use robotic control, the only differences being the size and sophistication. The mechanical principles, example program listings, and circuits you will use are very similar to, and sometimes the same as, industrial applications developed by engineers.

The goal of this text is to get students interested in and excited about the fields of engineering, mechatronics, and software development as they design, construct, and program an autonomous robot. This series of hands-on activities and projects will introduce students to basic robotic concepts using the Parallax Boe-Bot® robot, called the "Boe-Bot." Its name comes from the Board of Education® carrier board that is mounted on its wheeled chassis. An example of a Boe-Bot with an infrared obstacle detection circuit built on the Board of Education solderless prototyping area is shown below in Figure P-1.

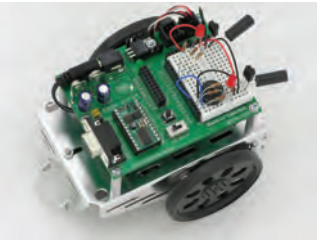


Figure P-1
Parallax Inc.’s
Boe-Bot® Robot

The activities and projects in this text begin with an introduction to your Boe-Bot’s brain, the Parallax BASIC Stamp® 2 microcontroller, and then move on to construction, testing,

and calibration of the Boe-Bot. After that, you will program the Boe-Bot for basic maneuvers, and then proceed to adding sensors and writing programs that make it react to its surroundings and perform autonomous tasks.

ABOUT VERSION 3.0

This is the first revision of this title since 2004. The major changes include:

- Replacement of the cadmium sulfide photoresistor with an RoHS-compliant light sensor of a type that will be more common in product design going forward. This required a rewrite of Chapter 6.
- Moving the "Setup and Testing" portion of Chapter 1 and the Hardware and Troubleshooting appendices to the Help file. This was done to support both serial and USB hardware connections, and other programming connections as our products and technologies continue to expand. This also allows for the dynamic maintenance of the Hardware and Troubleshooting material.
- Removal of references to the Parallax CD, which has been removed from our kits, reducing waste and ensuring that customers download the most recent BASIC Stamp Editor software and USB drivers available for their operating systems (www.parallax.com/go/Boe-Bot).

In addition, small errata items noted in the previous version (2.2) have been corrected. The material still aims for the same goals, and all of the same programming concepts and commands are covered, along with a few new ones. Finally, page numbers have been changed so the PDF page and the physical page numbers are the same, for ease of use.

AUDIENCE

This text is designed to be an entry point to technology literacy, and an easy learning curve for embedded programming and introductory robotics. The text is organized so that it can be used by the widest possible variety of students as well as independent learners. Middle-school students can try the examples in this text in a guided tour fashion by simply following the check-marked instructions with instructor supervision. At the other end of the spectrum, pre-engineering students' comprehension and problem-solving skills can be tested with the questions, exercises and projects (with solutions) in each chapter summary. The independent learner can work at his or her own pace, and obtain assistance through the Stamps in Class forum cited below.

SUPPORT FORUMS

Parallax maintains free, moderated forums for our customers, covering a variety of subjects:

- Propeller Chip: for all discussions related to the multicore Propeller microcontroller and development tools product line.
- BASIC Stamp: Project ideas, support, and related topics for all of the Parallax BASIC Stamp models.
- Sensors: Discussion relating to Parallax's wide array of sensors, and interfacing sensors with Parallax microcontrollers.
- Stamps in Class: Students, teachers, and customers discuss Parallax's education materials and school projects here.
- Robotics: For all Parallax robots and custom robots built with Parallax processors and sensors.
- Wireless: Topics include XBee, GSM/GPRS, telemetry and data communication over amateur radio.
- PropScope: Discussion and technical assistance for this USB oscilloscope that contains a Propeller chip.
- The Sandbox: Topics related to the use of Parallax products but not specific to the other forums.
- Projects: Post your in-process and completed projects here, made from Parallax products.

RESOURCES FOR EDUCATORS

We have a variety of resources for this text designed to support educators.

Stamps in Class "Mini Projects"

To supplement our texts, we provide a bank of projects for the classroom. Designed to engage students, each "Mini Project" contains full source code, "How it Works" explanations, schematics, and wiring diagrams or photos for a device a student might like to use. Many projects feature an introductory video, to promote self-study in those students most interested in electronics and programming. Just follow the Stamps in Class "Mini Projects" link at www.parallax.com/Education.

Educators Courses

These hands-on, intensive 1 or 2 day courses for instructors are taught by Parallax engineers or experienced teachers who are using Parallax educational materials in their classrooms. Visit www.parallax.com/Education → Educators Courses for details.

Parallax Educator's Forum

In this free, private forum, educators can ask questions and share their experiences with using Parallax products in their classrooms. Supplemental education materials are also posted here. To enroll, email education@parallax.com for instructions; proof of status as an educator will be required.

Supplemental Educational Materials

Select Parallax educational texts have an unpublished set of questions and solutions posted in our Parallax Educators Forum; we invite educators to copy and modify this material at will for the quick preparation of homework, quizzes, and tests. PowerPoint presentations and test materials prepared by other educators may be posted here as well.

Copyright Permissions for Educational Use

No site license is required for the download, duplication and installation of Parallax software for educational use with Parallax products on as many school or home computers as needed. Our Stamps in Class texts and BASIC Stamp Manual are all available as free PDF downloads, and may be duplicated as long as it is for educational use exclusively with Parallax microcontroller products and the student is charged no more than the cost of duplication. The PDF files are not locked, enabling selection of text and images to prepare handouts, transparencies, or PowerPoint presentations.

FOREIGN TRANSLATIONS

Many of our Stamps in Class texts have been translated into other languages; these texts are free downloads and subject to the same Copyright Permissions for Educational Use as our original versions. To see the full list, click on the Tutorials & Translations link at www.parallax.com/Education. These were prepared in coordination with the Parallax Volunteer Translator program. If you are interested in participating in our Volunteer Translator program, email translations@parallax.com.

ABOUT THE AUTHOR

Andy Lindsay joined Parallax Inc. in 1999, and has since authored eleven books and numerous articles and product documents for the company. The last three versions of *Robotics with the Boe-Bot* were 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 application and product engineering for Parallax.

SPECIAL CONTRIBUTORS

The Parallax team assembled to prepare this edition includes: excellent department leadership by Aristides Alvarez, lesson design and technical writing by Andy Lindsay; cover art by Jen Jacobs; graphic illustrations by Rich Allred and Andy Lindsay; nitpicking, editing, and layout by Stephanie Lindsay. Special thanks go to Ken Gracey, founder of the Stamps in Class program, and to Tracy Allen and Phil Pilgrim for consulting in the selection of the light sensor used in this version to replace the cadmium-sulfide photoresistor. Stephanie is particularly grateful to John Kauffman for his last-minute review of the revised Chapter 6.

1

Chapter 1: Your Boe-Bot's Brain

Parallax, Inc's Boe-Bot® robot is the focus of the activities, projects, and contests in this book. The Boe-Bot and a close-up of its BASIC Stamp® 2 programmable microcontroller brain are shown in Figure 1-1. The BASIC Stamp 2 module is both powerful and easy to use, especially with a robot.

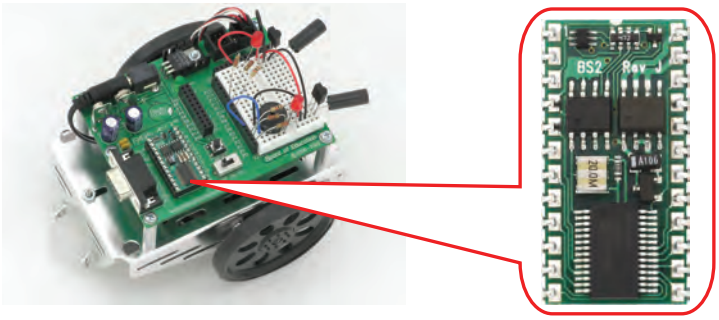


Figure 1-1
BASIC Stamp Module on a Boe-Bot Robot

The activities in this text will guide you through writing simple programs that make the BASIC Stamp and your Boe-Bot do four essential robotic tasks:

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

The programming language you will use to accomplish these tasks is called **PBASIC**, which stands for:

- Parallax - Company that invented and manufactures BASIC Stamp microcontrollers
- Beginners - Made for beginners to learn how to program computers
- All-purpose - Powerful and useful for solving many different kinds of problems
- Symbolic - Using symbols (terms that resemble English word/phrases)
- Instruction - To tell a computer what to do
- Code - In terms that the computer (and you) can understand

What's a Microcontroller? It's a programmable device that is designed into your digital wristwatch, cell phone, calculator, clock radio, etc. In these devices, the microcontroller has been programmed to sense when you press a button, make electronic beeping noises, and control the device's digital display. They are also built into factory machinery, cars, submarines, and spaceships because they can be programmed to read sensors, make decisions, and orchestrate devices that control moving parts.

The *What's a Microcontroller?* Student Guide is the recommended first text for beginners. It is full of examples of how to use microcontrollers, and how to make the BASIC Stamp the brain of your own microcontrolled inventions. It's available for free download from www.parallax.com/go/WAM, and it's also included in the BASIC Stamp Editor Help as a PDF file. It is included in the BASIC Stamp Activity Kit and BASIC Stamp Discovery Kit, which are carried by many electronic retailers. These kits can also be purchased directly from Parallax, either online at www.parallax.com/go/WAM or by phone at (888) 512-1024.

HARDWARE AND SOFTWARE

Getting started with BASIC Stamp microcontroller modules is similar to getting started with a brand-new PC or laptop. The first things that most people have to do is take it out of the box, plug it in, install and test some software, and maybe even write some software of their own using a programming language. If this is your first time using a BASIC Stamp module, you will be doing all these same activities. If you are in a class, your hardware may already be all set up for you. If this is the case, your teacher may have other instructions. If not, this chapter will take you through all the steps of getting your new BASIC Stamp microcontroller up and running.

ACTIVITY #1: GETTING THE SOFTWARE

The BASIC Stamp Editor (version 2.5 or higher) is the software you will use in most of the activities and projects in this text. You will use this software to write programs that the BASIC Stamp module will run. You can also use this software to display messages sent by the BASIC Stamp that help you understand what it senses.

Computer System Requirements

You will need a personal computer to run the BASIC Stamp Editor software. Your computer will need to have the following features:

- Microsoft Windows 2K/XP/Vista/7 or newer operating system
- An available serial or USB port
- Internet access and an Internet browser program

Downloading the Software from the Internet

It is important to always use the latest version of the BASIC Stamp Editor software if possible. The first step is to go to the Parallax web site and download the software.

- ✓ Using a web browser, go to www.parallax.com/basicstampsoftware.

Figure 1-2: BASIC Stamp Editor download page at www.parallax.com/basicstampsoftware

Use the "Click Here to Download" button to get the latest version of the software.

- ✓ Click on the Click Here to Download button to download the latest version of the BASIC Stamp Windows Editor software.

✓ A File Download window will open, asking you if you want to run or to save this file (Figure 1-3). Click Save.

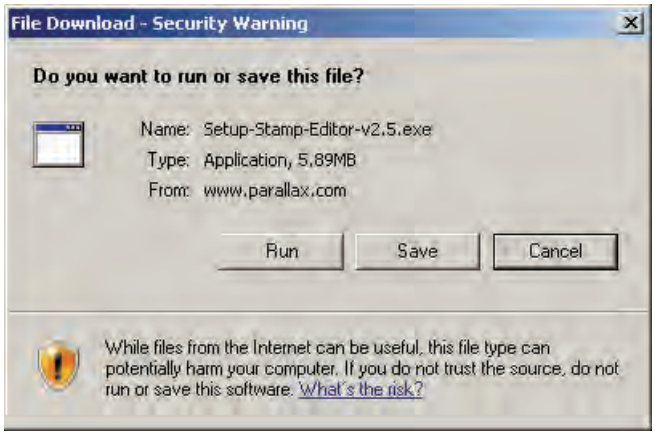


Figure 1-3
File Download Window

Click Save, then save the file to your computer.

✓ Follow the prompts that appear. When the download is complete, click Run. You may see messages from your operating system asking you to verify that you wish to continue with installation. Always agree that you want to continue.

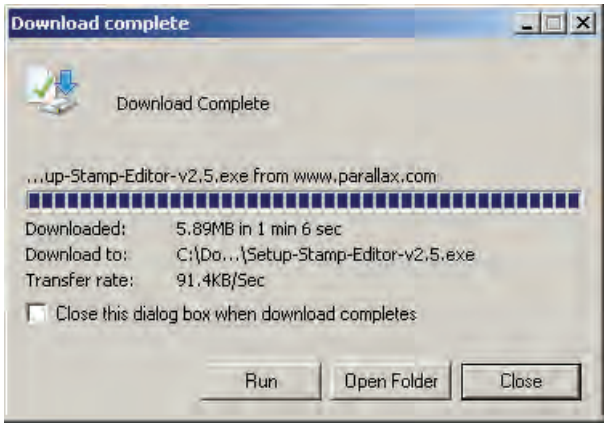


Figure 1-4
Download Complete Message

Click Run.

If prompted, always confirm you want to continue.

✓ The BASIC Stamp Editor Installer window will open (Figure 1-5). Click Next and follow the prompts, accepting all defaults.

Figure 1-5
BASIC Stamp Editor Installer Window

Click Next.

✓ IMPORTANT: When the "Install USB Driver" message appears (Figure 1-6), leave the checkmark in place for the Automatically install/update driver (recommended) box, and then click Next.

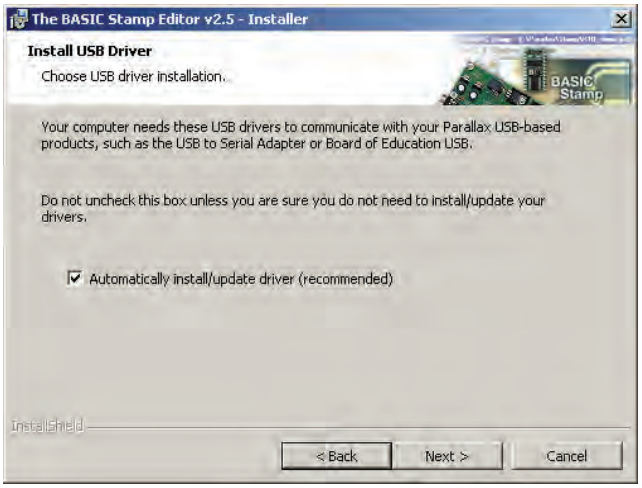


Figure 1-6
Install USB Driver Message

Leave the box checked, and click Next.

- ✓ When the "Ready to Install the Program" message appears, click the Install button. A progress bar may appear, and this could take a few minutes.

At this point, an additional window may appear behind the current window while the USB drivers are updating. This window will eventually close on its own when the driver installation is complete. If you don't see this window, it does not indicate a problem.

About USB drivers. The USB drivers that install with the BASIC Stamp Windows Editor installer by default are necessary to use any Parallax hardware connected to your computer's USB port. VCP stands for Virtual COM Port, and it will allow your computer's USB port to look and be treated as a standard RS232 serial port by Parallax hardware.

USB Drivers for Different Operating Systems The USB VCP drivers included in the BASIC Stamp Windows Editor software are for certain Windows operating systems only. For more information, visit www.parallax.com/usbdrivers.

- ✓ When the window tells you that installation has been successfully completed, click Finish (Figure 1-7).

Figure 1-7
BASIC Stamp Editor Installation Completed

Click Finish.

ACTIVITY #2: USING THE HELP FILE FOR HARDWARE SETUP

In this section you will run the BASIC Stamp Editor's Help file. Within the Help file, you will learn about the different BASIC Stamp programming boards available for the Stamps in Class program, and determine which one you are using. Then, you will follow the steps in the Help to connect your hardware to your computer and test your BASIC Stamp programming system.

Running the BASIC Stamp Editor for the first time

- ✓ If you see the BASIC Stamp Editor icon on your computer desktop, double-click it (Figure 1-8).
- ✓ Or, click on your computer's Start menu, then choose All Programs ▸ Parallax Inc ▸ BASIC Stamp Editor 2.5 ▸ BASIC Stamp Editor 2.5.

Figure 1-8
BASIC Stamp Editor Desktop Icon

Double-click to launch the program.

- ✓ On the BASIC Stamp Editor's toolbar, click Help on the toolbar (Figure 1-9) and then select BASIC Stamp Help… from the drop-down menu.

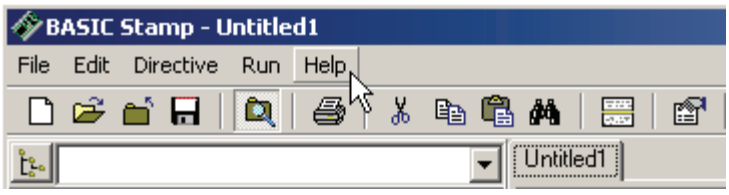
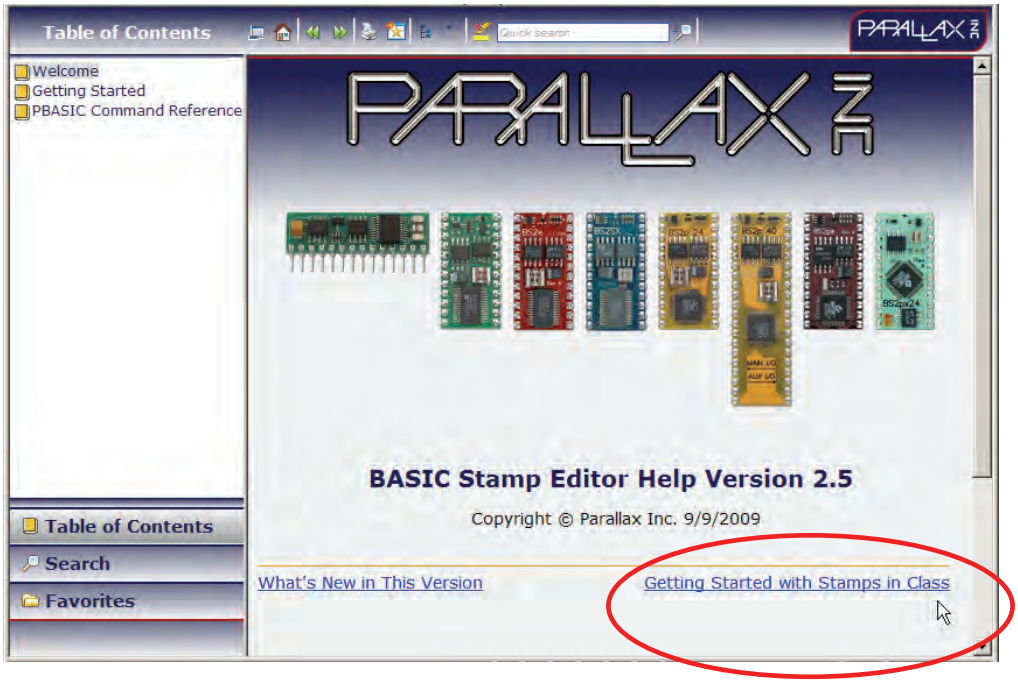


Figure 1-9
Opening the Help Menu

Click Help, then choose BASIC Stamp Help from the drop-down menu.

Figure 1-10: BASIC Stamp Editor Help



✓ Click on the Getting Started with Stamps in Class link on the bottom of the Welcome page, as shown in the lower right corner of Figure 1-10.

Following the Directions in the Help File

From here, you will follow the directions in the Help file to complete these tasks:

- Identify which BASIC Stamp development board you are using
- Connect your development board to your computer
- Test your programming connection
- Troubleshoot your programming connection, if necessary
- Write your first PBASIC program for your BASIC Stamp
- Power down your hardware when you are done

When you have completed the activities in the Help file, return to this book and continue with the Summary below before moving on to Chapter 2.

What do I do if I get stuck?

If you run into problems while following the directions in this book or in the Help file, you have many options to obtain free Technical Support:

- **Forums:** sign up and post a message in our free, moderated Stamps in Class forum at forums.parallax.com.
- **Email:** send an email to support@parallax.com.
- **Telephone:** In the Continental United States, call toll-free to 888-99-STAMP (888-997-8267). All others call (916) 624-8333.
- **More resources:** Visit www.parallax.com/support.

SUMMARY

This chapter guided you through the following:

- An introduction to the BASIC Stamp module
- Where to get the free BASIC Stamp Editor software you will use in just about all of the experiments in this text
- How to install the BASIC Stamp Editor software
- How to use the BASIC Stamp Editor's Help and the BASIC Stamp Manual
- An introduction to the BASIC Stamp module, Board of Education, and HomeWork Board
- How to set up your BASIC Stamp hardware
- How to test your software and hardware
- How to write and run a PBASIC program
- Using the **`DEBUG`** and **`END`** commands, **`CR`** control character, and **`DEC`** formatter.

- A brief introduction to ASCII code
- How to disconnect the power to your Board of Education or HomeWork Board when you're done

Questions

1. What device will be the brain of your Boe-Bot?
2. When the BASIC Stamp sends a character to your PC/laptop, what type of numbers are used to send the message through the programming cable?
3. What is the name of the window that displays messages sent from the BASIC Stamp to your PC/laptop?
4. What PBASIC commands did you learn in this chapter?

Exercises

1. Explain what the asterisk does in this command: **`DEBUG DEC 7 * 11`**

2. Guess what the Debug Terminal would display if you ran this command: **`DEBUG DEC 7 + 11`**

3. There is a problem with these two commands. When you run the code, the numbers they display are stuck together so that it looks like one large number instead of two small ones. Modify these two commands so that the answers appear on different lines in the Debug Terminal.

```
DEBUG DEC 7 * 11
DEBUG DEC 7 + 11
```

Projects

1. Use **`DEBUG`** to display the solution to the math problem: 1 + 2 + 3 + 4.
2. Save FirstProgramYourTurn.bs2 under another name. If you were to place the **`DEBUG`** command shown below on the line just before the **`END`** command in the program, what other lines could you delete and still have it work the same? Modify the copy of the program to test your hypothesis (your prediction of what will happen).

```
DEBUG "What's 7 X 11?", CR, "The answer is: ", DEC 7 * 11
```

Solutions

Q1. A BASIC Stamp 2 microcontroller module.
Q2. Binary numbers, that is, 0's and 1's.
Q3. The Debug Terminal.
Q4. **DEBUG** and **END**
E1. It multiplies the two operands 7 and 11, resulting in a product of 77. The asterisk is the multiply operator.
E2. The Debug Terminal would display: 18
E3. To fix the problem, add a carriage return using the **CR** control character and a comma.

```
DEBUG DEC 7 * 11
DEBUG CR, DEC 7 + 11
```

P1. Here is a program to display a solution to the math problem: 1+2+3+4.

```
' What's a Microcontroller - Ch01Prj01_Add1234.bs2
'{$STAMP BS2}
'{$PBASIC 2.5}

DEBUG "What's 1+2+3+4?"
DEBUG CR, "The answer is: "
DEBUG DEC 1+2+3+4

END
```

P2. The last three **DEBUG** lines can be deleted. An additional **CR** is needed after the "Hello" message.

```
' What's a Microcontroller - Ch01Prj02_ FirstProgramYourTurn.bs2
' BASIC Stamp sends message to Debug Terminal.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Hello, it's me, your BASIC Stamp!", CR
DEBUG "What's 7 X 11?", CR, "The answer is: ", DEC 7 * 11

END
```

The output from the Debug Terminal is:

```
Hello, it's me, your BASIC Stamp!
What's 7 X 11?
The answer is: 77
```

This output is the same as it was with the previous code. This is an example of using commas to output a lot of information, using only one **`DEBUG`** command with multiple elements in it.

Chapter 2: Your Boe-Bot's Servo Motors

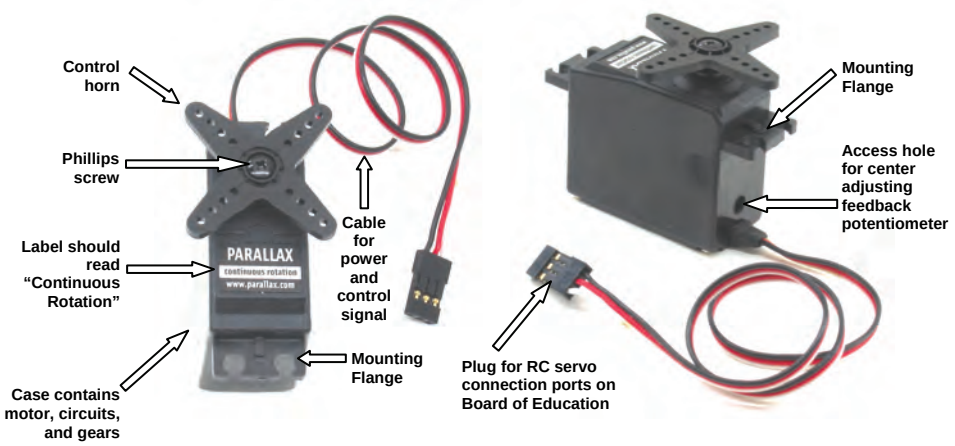
2

This chapter will guide you through connecting, adjusting, and testing the Boe-Bot's motors. In order to do that, you will need to understand certain PBASIC commands and programming techniques that will control the direction, speed, and duration of servo motions. Therefore, Activities #1, #2, and #5 will introduce you to these programming tools, and then Activities #3, #4, and #6 will show you how to apply them to the servos. Since precise servo control is key to the Boe-Bot's performance, completing these activities before mounting the servos into the Boe-Bot chassis is both important and necessary!

INTRODUCING THE CONTINUOUS ROTATION SERVO

The Parallax Continuous Rotation servos shown in Figure 2-1 are the motors that will make the Boe-Bot's wheels turn. This figure points out the servos' external parts. Many of these parts will be referred to as you go through the instructions in this and the next chapter.

Figure 2-1 Parallax Continuous Rotation Servo



Note: You might find it useful to bookmark this page so that you can refer back to it later.

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 of holding certain positions, they turn at certain speeds and directions. Continuous rotation servos are ideal for controlling wheels and pulleys.

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-Bot. You will be removing the control horn with a wheel.

ACTIVITY #1: BUILDING AND TESTING THE LED CIRCUIT

Controlling a servo motor's speed and direction involves a program that makes the BASIC Stamp send the same message, over and over again. The message has to repeat itself around 50 times per second for the servo to maintain its speed and direction. This activity has a few PBASIC example programs that demonstrate how to repeat the same message over and over again and control the timing of the message.

Displaying Messages at Human Speeds

You can use the **PAUSE** command to tell the BASIC Stamp to wait for a while before executing the next command.

PAUSE *Duration*

The number that you put to the right of the **PAUSE** command is called the ***Duration*** argument, and it's the value that tells the BASIC Stamp how long it should wait before moving on to the next command. The units for the ***Duration*** argument are thousandths of a second (ms). So, if you want to wait for one second, use a value of 1000. Here's how the command should look:

```
PAUSE 1000
```

If you want to wait for twice as long, try:

```
PAUSE 2000
```

A second is abbreviated "s." In this text, when you see 1 s, it means one second.

A millisecond is one thousandth of a second, and it is abbreviated "ms." The command **PAUSE 1000** delays the program for 1000 ms, which is 1000/1000 of a second, which is one second, or 1 s. Got it?

Example Program: TimedMessages.bs2

There are lots of different ways to use the **PAUSE** command. This example program uses **PAUSE** to delay between printing messages that tell you how much time has elapsed. The program should wait one second before it sends the "One second elapsed…" message and another two seconds before it displays the "Three seconds elapsed . . . " message.

- ✓ If you have a Board of Education, move the 3-postion switch from position-0 to position-1.
- ✓ If you have a HomeWork Board, reconnect the 9 V battery to the battery clip.
- ✓ Enter the program below into the BASIC Stamp Editor.
- ✓ Save the program under the name TimedMessages.bs2.
- ✓ Run the program, and then watch for the delay between messages.

```
' Robotics with the Boe-Bot - TimedMessages.bs2
' Show how the PAUSE command can be used to display messages at human speeds.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Start timer..."

PAUSE 1000
DEBUG CR, "One second elapsed..."

PAUSE 2000
DEBUG CR, "Three seconds elapsed..."

DEBUG CR, "Done."

END
```

From here onward, the three instructions that came before this program will be phrased like this:

- ✓ Enter, save, and run TimedMessages.bs2.

Your Turn – Different Pause Durations

You can change the delay between messages by changing the **PAUSE** commands' ***Duration*** arguments.

- ✓ Try changing the **PAUSE** ***Duration*** arguments from 1000 and 2000 to 5000 and 10000, for example:

```
DEBUG "Start timer..."

PAUSE 5000
DEBUG CR, "Five seconds elapsed..."

PAUSE 10000
DEBUG CR, "Fifteen seconds elapsed..."
```

- ✓ Run the modified program.
- ✓ Also try it again with numbers like 40 and 100 for the ***Duration*** arguments; they'll go pretty fast.
- ✓ The longest possible ***Duration*** argument is 65535. If you've got a minute to spare, try **PAUSE 60000**.

Over and Over Again

One of the best things about both computers and microcontrollers is that they never complain about doing the same boring things over and over again. You can place your commands between the words **DO** and **LOOP** if you want them executed over and over again. For example, let's say you want to print a message repeating once every second. Simply place your **DEBUG** and **PAUSE** commands between the words **DO** and **LOOP** like this:

```
DO
  DEBUG "Hello!", CR
  PAUSE 1000
LOOP
```

Example Program: HelloOnceEverySecond.bs2

- ✓ Enter, save, and run HelloOnceEverySecond.bs2.
- ✓ Verify that the "Hello!" message is printed once every second.

```
' Robotics with the Boe-Bot - HelloOnceEverySecond.bs2
' Display a message once every second.

' {$STAMP BS2}
' {$PBASIC 2.5}

DO
  DEBUG "Hello!", CR
  PAUSE 1000
LOOP
```

Your Turn – A Different Message

You can modify your program so that part of it executes once, and another part executes over an over again.

- ✓ Modify the program so that the commands look like this:

```
DEBUG "Hello!"
DO
  DEBUG "!"
  PAUSE 1000
LOOP
```

- ✓ Run it and see what happens! Did you anticipate the result?

ACTIVITY #2: TRACKING TIME AND REPEATING ACTIONS WITH A CIRCUIT

In this activity, you will build circuits that emit light that will allow you to "see" the kind of signals that are used to control the Boe-Bot's servo motors.

What's a Microcontroller? This activity contains selected excerpts from the *What's a Microcontroller?* Student Guide.

- ✓ Even if you are familiar with this material from *What's a Microcontroller?*, don't skip this activity.

In the second half of this activity, you will examine the signals that control your servos and timing diagrams in a different light than they were presented in *What's a Microcontroller?*

Bonus! The components in your Boe-Bot kit can be used to complete many of the activities in *What's a Microcontroller?* Go www.parallax.com/go/WAM for a complete list, and to download the text.

Introducing the LED and 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, and the sign for the ohm is the Greek letter omega: Ω. The resistor you will be working with in this activity is the 470 Ω resistor shown in Figure 2-2. The resistor has two wires (called leads and pronounced "leeds"), one coming out of each end. There is a ceramic case between the two leads, and it's the part that resists current flow. Most circuit diagrams that show resistors use the symbol on the left with the squiggly lines to tell the person building the circuit that he or she must use a 470 Ω resistor. This is called a schematic symbol. The drawing on the right is a part drawing used in some beginner level Stamps in Class texts to help you build circuits.

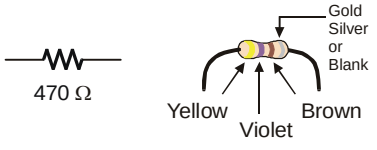


Figure 2-2
470 Ω Resistor Part Drawing

Schematic symbol (left) and Part Drawing (right)

The colored stripes indicate resistance values. See Appendix B: Resistor Color Codes and Breadboarding Rules on page 293 for information on how to determine a resistor's value from the colored stripes on its ceramic case.

A diode is a one-way current valve, and a light-emitting diode (LED) emits light when current passes through it. Unlike the color codes on a resistor, the color of the LED usually just tells you what color it will glow when current passes through it. The important markings on an LED are contained in its shape. Since an LED is a one-way current valve, you have to make sure to connect it the right way, or it won't work as intended.

Figure 2-3 shows an LED's schematic symbol and part drawing. An LED has two terminals. One is called the anode, and the other is called the cathode. In this activity, you will have to build the LED into a circuit, and you will have to pay attention and make sure the anode and cathode leads are connected to the circuit properly. On the part drawing, the anode lead is labeled with the plus-sign (+). On the schematic symbol, the anode is the wide part of the triangle. In this part drawing, the cathode lead is the pin

labeled with a minus-sign (-), and on the schematic symbol, the cathode is the line across the point of the triangle.

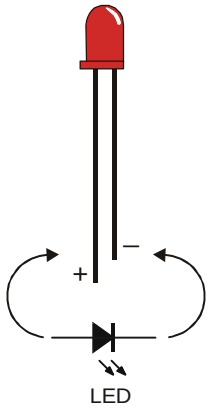


Figure 2-3
LED Part Drawing and Schematic Symbol

Part drawing (above) and schematic symbol (below)

The LED part drawings in later pictures will have a + next to the anode leg.

When you start building your circuit, make sure to check it against the schematic symbol and part drawing. If you look closely at the LED's plastic case in the part drawing, it's mostly round, but there is a small flat spot right near one of the leads that that tells you it's the cathode. Also note that the LED's leads are different lengths. In this text, the anode will be shown with a (+) sign and the cathode will be shown with a (–) sign.

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. Therefore, it is 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 will not light up.

LED Test Circuit Parts

(2) LEDs – Red
(2) Resistors, 470 Ω (yellow-violet-brown)

Always disconnect power to your board before building or modifying circuits! For the Board of Education, set the 3-position switch to position-0. For the BASIC Stamp HomeWork Board, disconnect the 9 V battery from the battery clip. Always double-check your circuit for errors before reconnecting power.

LED Test Circuits

If you completed the *What's a Microcontroller?* text, you are no doubt very familiar with the circuit shown in Figure 2-4. The left side of this figure shows the circuit schematic, and the right side shows a wiring diagram example of the circuit built on your board's prototyping area.

- ✓ Build the circuit shown in Figure 2-4.
- ✓ Make sure that the shorter pins on each LED (the cathodes) are plugged into black sockets labeled Vss.
- ✓ Make sure the longer pins (the anodes, marked with a ⊕ in the wiring diagram) are connected to the white breadboard sockets exactly as shown.

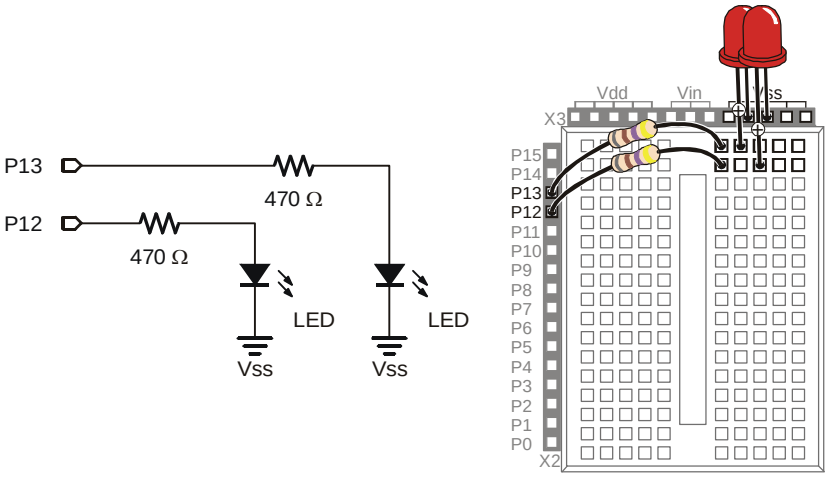


Figure 2-4
Two LEDs Connected to BASIC Stamp I/O Pins P13 and P12

Schematic (left) and wiring diagram (right)

What's an I/O pin? I/O stands for input/output. The BASIC Stamp 2 has 24 pins, 16 of which are I/O pins. In this text, you will program the BASIC Stamp to use I/O pins as outputs to make LED lights turn on/off, control the speed and direction the Parallax Continuous Rotation servos turn, make tones with a speaker, and prepare sensors to detect light and objects. You will also program the BASIC Stamp to use I/O pins as inputs to monitor sensors that indicate mechanical contact, light level, objects in the Boe-Bot's path, and even their distance.

New to building circuits? See Appendix B: Resistor Color Codes and Breadboarding Rules on page 293.

Figure 2-5 shows what you will program the BASIC Stamp to do to the LED circuit. Imagine that you have a 5 volt (5 V) battery. Although a 5 V battery is not common, the Board of Education has a device called a voltage regulator that supplies the BASIC Stamp with the equivalent of a 5 V battery. When you connect a circuit to Vss, it's like connecting the circuit to the negative terminal of the 5 V battery. When you connect the other end of the circuit to Vdd, it's like connecting it to the positive terminal of a 5 V battery.

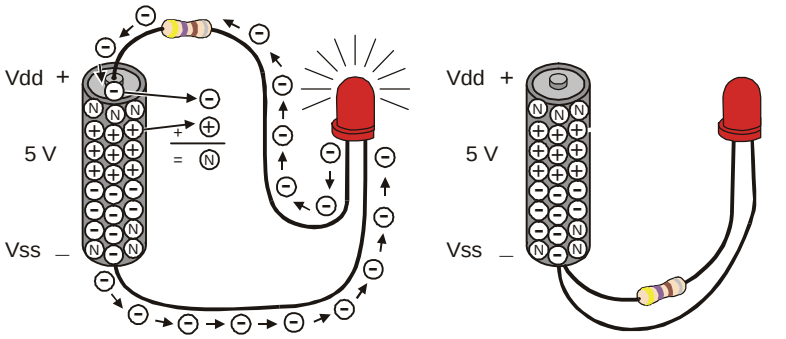


Figure 2-5
Current and Electron Flow

***Volts* is abbreviated V.** That means 5 volts is abbreviated 5 V. When you apply voltage to a circuit, it's like applying electrical pressure.

***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 amount of current an electric motor draws is often measured in amps, for example 2 A, 5 A, etc. However, the currents you will use in the Board of Education are measured in thousandths of an amp, or milliamps. For example, 10.3 mA passes through the circuit in Figure 2-5.

When these connections are made, 5 V of electrical pressure is applied to the circuit causing electrons to flow through and the LED to emit light. As soon as you disconnect the resistor lead from the battery's positive terminal, the current stops flowing, and the LED stops emitting light. You can take it one step further by connecting the resistor lead to Vss, which has the same result. This is the action you will program the BASIC Stamp to do to make the LED turn on (emit light) and off (not emit light).

Programs that Control the LED Test Circuits

The **HIGH** and **LOW** commands can be used to make the BASIC Stamp connect an LED alternately to Vdd and Vss. The ***Pin*** argument is a number between 0 and 15 that tells the BASIC Stamp which I/O pin to connect to Vdd or Vss.

HIGH *Pin*
LOW *Pin*

For example, if you use the command:

```
HIGH 13
```

...it tells the BASIC Stamp to connect I/O pin P13 to Vdd, which turns the LED on.

Likewise, if you use the command

```
LOW 13
```

...it tells the BASIC Stamp to connect I/O pin P13 to Vss, which turns the LED off. Let's try this out.

Example Program: HighLowLed.bs2

- ✓ Reconnect power to your board.
- ✓ Enter, save, and run HighLowLed.bs2.
- ✓ Verify that the LED circuit connected to P13 is turning on and off, once every second.

```
' Robotics with the Boe-Bot – HighLowLed.bs2
' Turn the LED connected to P13 on/off once every second.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "The LED connected to Pin 13 is blinking!"

DO

  HIGH 13
  PAUSE 500
  LOW 13
  PAUSE 500

LOOP
```

How HighLowLed.bs2 Works

Figure 2-6 shows how the BASIC Stamp can connect an LED circuit alternately to Vdd and Vss. When it's connected to Vdd, the LED emits light. When it's connected to Vss, the LED does not emit light. The command **`HIGH 13`** instructs the BASIC Stamp to connect P13 to Vdd. The command **`PAUSE 500`** instructs the BASIC Stamp to leave the circuit in that state for 500 ms. The command **`LOW 13`** instructs the BASIC Stamp to connect the LED to Vss. Again, the command **`PAUSE 500`** instructs the BASIC Stamp to leave it in that state for another 500 ms. Since these commands are placed between **`DO`** and **`LOOP`**, they execute over and over again.

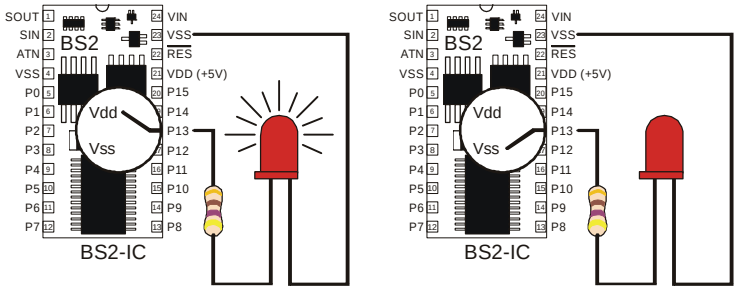


Figure 2-6
BASIC Stamp Switching

The BASIC Stamp can be programmed to internally connect the LED circuit's input to Vdd or Vss.

A Diagnostic Test for your Computer

A very few computers, such as some laptops, will halt the PBASIC program after the first time through a **`DO...LOOP`** instruction. These computers have a non-standard serial port design. By placing a **`DEBUG`** command the program LedOnOff.bs2, the open Debug Terminal prevents this from possibly happening. You will next re-run this program without the **`DEBUG`** command to see if your computer has this non-standard serial port problem. It is not likely, but it would be important for you to know.

- ✓ Open HighLowLed.bs2.
- ✓ Delete the entire **`DEBUG`** instruction.
- ✓ Run the modified program while you observe your LED.

If the LED blinks on and off continuously, just as it did when you ran the original program with the **`DEBUG`** command, your computer will not have this problem.

If the LED blinked on and off only once and then stopped, you have a computer with a non-standard serial port design. If you disconnect the programming cable from your board and press the Reset button, the BASIC Stamp will run the program properly without freezing. In programs you write yourself, you should add a single command:

```
DEBUG "Program Running!"
```

...right after the compiler directives. This will open the Debug Terminal and keep the COM port open. This will prevent your programs from freezing after one pass through the **DO…LOOP**, or any of the other looping commands you will be learning in later chapters. You will see this command in some of the example programs that would not otherwise need a **DEBUG** instruction. So, you should be able to run all of the remaining programs in this book even if your computer failed the diagnostic test.

Introducing the Timing Diagram

A timing diagram is a graph that relates high (Vdd) and low (Vss) signals to time. In Figure 2-7, time increases from left to right, and high and low signals align with either Vdd (5 V) or Vss (0 V). This timing diagram shows you a 1000 ms slice of the high/low signal you just experimented with. The line of dots (. . .) to the right of the signal is one way of indicating that the signal repeats itself.

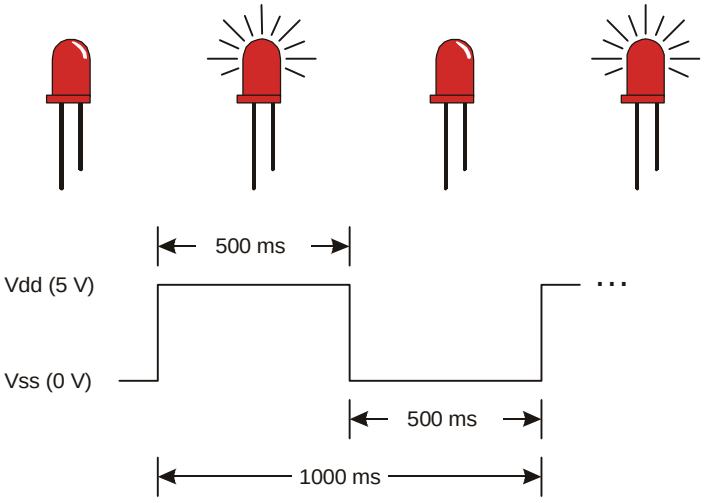


Figure 2-7
Timing Diagram for HighLowLed.bs2

The LED on/off states are shown above the timing diagram.

Your Turn – Blink the Other LED

Blinking the other LED (connected to P12) is a simple matter of changing the ***Pin*** argument in the **HIGH** and **LOW** commands and re-running the program.

- ✓ Modify the program so that the commands look like this:

```
DO
  HIGH 12
  PAUSE 500
  LOW 12
  PAUSE 500
LOOP
```

- ✓ Run the modified program and verify that it makes the other LED blink on/off.

You can also make both LEDs blink at the same time.

- ✓ Modify the program so that the commands look like this:

```
DO
  HIGH 12
  HIGH 13
  PAUSE 500
  LOW 12
  LOW 13
  PAUSE 500
LOOP
```

- ✓ Run the modified program and verify that it makes both LEDs blink on and off at roughly the same time.

You can modify the program again to make one LEDs blink alternately on/off, and you can also change the rates that the LEDs blink by adjusting the **PAUSE** command's ***Duration*** argument higher or lower.

- ✓ Try it!

Viewing a Servo Control Signal with an LED

The high and low signals you will program the BASIC Stamp to send to the servo motors must last for very precise amounts of time. That's because the servo motors measure the amount of time the signal stays high, and use it as an instruction for where to turn. For accurate servo motor control, the time these signals stay high must be much more precise than you can get with a **HIGH** and a **PAUSE** command. You can only change the **PAUSE** command's ***Duration*** argument by 1 ms (remember, that's 1/1000 of a second) at a time. There's a different command called **PULSOUT** that can deliver high signals for precise amounts of time. These amounts of time are values you use in the ***Duration*** argument, and they are measured in units that are two millionths of a second!

PULSOUT *Pin*, *Duration*

A microsecond is a millionth of a second. It's abbreviated µs. Be careful when you write this value, it's not the letter 'u' from our alphabet; it's the Greek letter mu 'µ'.
For example, 8 microseconds is abbreviated 8 µs.

You can send a **HIGH** signal that turns the P13 LED on for 2 µs (that's two millionths of a second) by using this command:

```
PULSOUT 13, 1
```

This command would turn the LED on for 4 µs:

```
PULSOUT 13, 2
```

This command sends a high signal that you can actually view:

```
PULSOUT 13, 65000
```

How long does the LED circuit connected to P13 stay on when you send this pulse? Let's figure it out. The time it stays on is 65000 times 2 µs. That's:

$$\begin{aligned} Duration &= 65000 \times 2\,\mu s \\ &= 65000 \times 0.000002\,s \\ &= 0.13\,s \end{aligned}$$

...which is still pretty fast, thirteen hundredths of a second.

The largest value you can use in a PULSOUT *Duration* argument is 65535.

Example Program: PulseP13Led.bs2

This timing diagram in Figure 2-8 shows the pulse train you are about to send to the LED with this new program. This time, the high signal lasts for 0.13 seconds, and the low signal lasts for 2 seconds. This is 100 times slower than the signal that the servo will need to control its motion.

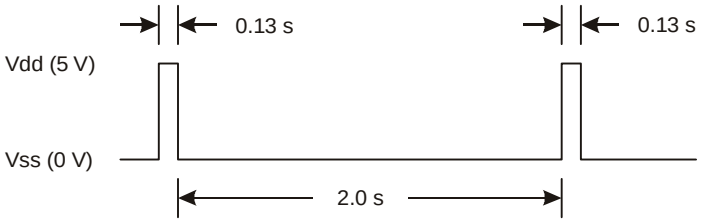


Figure 2-8
Timing Diagram for PulseP13Led.bs2

- ✓ Enter, save, and run PulseP13Led.bs2.
- ✓ Verify that the LED circuit connected to P13 pulses for about thirteen hundredths of a second, once every two seconds.

```
' Robotics with the Boe-Bot – PulseP13Led.bs2
' Send a 0.13 second pulse to the LED circuit connected to P13 every 2 s.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

DO

  PULSOUT 13, 65000
  PAUSE 2000

LOOP
```

Example Program: PulseBothLeds.bs2

This example program sends a pulse to the LED connected to P13, and then it sends a pulse to the LED connected to P12 as shown in Figure 2-9. After that, it pauses for two seconds.

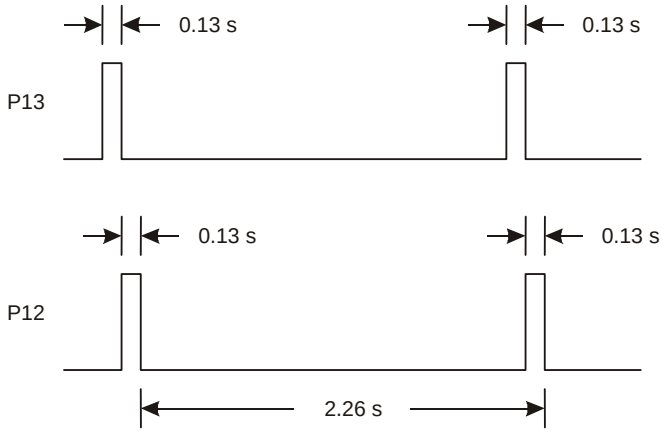


Figure 2-9
Timing Diagram for PulseBothLeds.bs2

The LEDs emit light for 0.13 second while the signal is high.

The voltages (Vdd and Vss) in this timing diagram are not labeled. With the BASIC Stamp, it is understood that the high signal is 5 V (Vdd) and the low signal is 0 V (Vss).

This is a common practice in documents that explain the timing of high and low signals. Often there are one or more of these documents for each component inside the circuit an engineer is designing. The engineers who created the BASIC Stamp had to comb through many of these kinds of documents looking for information needed to help make decisions while designing the product.

Sometimes the times are also left out, or just shown with a label, like t_{high} and t_{low}. Then, the desired time values for t_{high} and t_{low} are listed in a table somewhere after the timing diagram. This concept is discussed in more detail in *Basic Analog and Digital,* another Parallax Stamps in Class Student Guide.

- ✓ Enter, save, and run PulseBothLeds.bs2.
- ✓ Verify that both LED circuits simultaneously pulse for about thirteen hundredths of a second, once every two seconds.

```
' Robotics with the Boe-Bot – PulseBothLeds.bs2
' Send a 0.13 second pulse to P13 and P12 every 2 seconds.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

DO
  PULSOUT 13, 65000
  PULSOUT 12, 65000
  PAUSE 2000
LOOP
```

Your Turn – Viewing the Full Speed Servo Signal

Remember the servo signal is 100 times as fast as the program you just ran. First, let's try running the program ten times as fast. That means divide all the ***Duration*** arguments (**PULSOUT** and **PAUSE**) by 10.

- ✓ Modify the program so that the commands look like this:

```
DO
  PULSOUT 13, 6500
  PULSOUT 12, 6500
  PAUSE 200
LOOP
```

- ✓ Run it and verify that it makes the LEDs blink ten times as fast.

Now, let's try 100 times as fast (one hundredth of the duration). Instead of appearing to flicker, the LED will just appear to be not as bright as it would when you send it a simple high signal. That's because the LED is flashing on and off so quickly and for such brief periods of time that the human eye cannot detect the actual on/off flicker, just a change in brightness.

- ✓ Modify the program so that the commands look like this:

```
DO
  PULSOUT 13, 650
  PULSOUT 12, 650
  PAUSE 20
LOOP
```

- ✓ Run the modified program and verify that it makes both LEDs about the same brightness.
- ✓ Try substituting 850 in the ***Duration*** argument for the P13 **PULSOUT** command.

```
DO
  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20
LOOP
```

- ✓ Run the modified program and verify that the P13 LED now appears slightly brighter than the P12 LED. You may have to cup your hands around the LEDs and peek inside to see the difference. They differ because the amount of time the P13 LED stays on is longer than the amount of time the P12 LED stays on.

- ✓ Try substituting 750 in the ***Duration*** argument for both the **PULSOUT** commands.

```
DO
  PULSOUT 13, 750
  PULSOUT 12, 750
  PAUSE 20
LOOP
```

- ✓ Run the modified program and verify that the brightness of both LEDs is the same again. It may not be obvious, but the brightness level is between those given by ***Duration*** arguments of 650 and 850.

ACTIVITY #3: CONNECTING THE SERVO MOTORS

In this activity, you will build a circuit that connects the servo to a power supply and a BASIC Stamp I/O pin. The LED circuits you developed in the last activity will be used later to monitor the signals the BASIC Stamp sends to the servos to control their motion.

Parts for Connecting the Servos

(2) Parallax Continuous Rotation servos
(2) Built and tested LED circuits from the previous activity

Finding the Connection Instructions for Your Carrier Board

There are different revisions of the Board of Education and BASIC Stamp HomeWork Board. Furthermore, there are several variations to the Board of Education, based on programming interface. In Chapter 1, you used the BASIC Stamp Editor Help file to determine the type and revision of your board, and special instructions for older boards.

The instructions in this book were written to support the boards that were current at the time of writing, and previous compatible revisions:

- Board of Education Serial - Rev C or higher
- Board of Education USB - Rev A or higher
- BASIC Stamp HomeWork Board Serial - Rev C or higher

✓ Examine the labeling on your carrier board and make note of the type and the revision.
✓ For older boards, check the BASIC Stamp Editor Help for notes specific to your board.

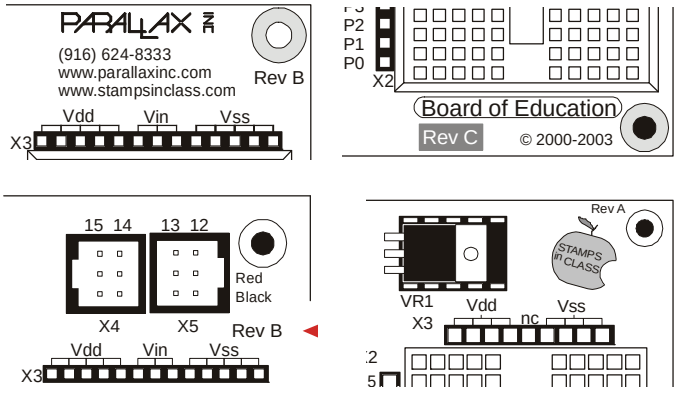


Figure 2-10
BASIC Stamp Switching

The BASIC Stamp can be programmed to internally connect the LED circuit's input to Vdd or Vss.

✓ If your board is one of the type and revisions listed above, go to one of the following pages to continue:

- Board of Education: go to page 42.
- HomeWork Board: go to page 45.

Connecting the Servos to the Board of Education

- ✓ Turn off the power by setting the 3-position switch on your Board of Education to position-0 (see Figure 2-11).

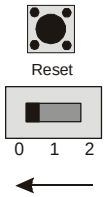


Figure 2-11
Turn Off Power

Figure 2-12 shows the servo header on the Board of Education. This board features a jumper that you can use to connect the servo's power supply to either Vin or Vdd. To move it, you have to pull it upwards and off the pair of pins it rests on, then push it onto the pair of pins you want it to rest on.

- ✓ If you are using the 6 V battery pack, make sure the jumper between the servo ports on the Board of Education is set to Vin as shown on the left of Figure 2-12.

About Rechargeable Batteries. The Boe-Bot requires 6 V, easily obtained from 4 AA 1.5 V batteries. Alkaline AA batteries are 1.5 V. However, many rechargeable AA batteries supply only 1.2 V, giving a total of 4.8 V, which is not enough to power the BASIC Stamp and Boe-Bot. If you cannot find 1.5 V rechargeable batteries, you may use the inexpensive Boe-Boost (#30078) to add a 5th 1.2 V rechargeable battery, bringing the total back to 6 V.

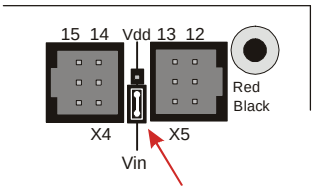
- ✓ If you are using a 7.5 V, 1000 mA center positive DC supply, set the jumper to Vdd as shown on the right side of Figure 2-12.

CAUTION – Misuse of AC powered DC supplies can damage your servos.

If you are inexperienced with DC supplies, consider sticking with the 6 V battery pack that comes with the Boe-Bot.

Use only supplies with DC output voltage ratings between 6 and 7.5 V, and current output ratings of 800 mA or more.

Only use a DC supply that is equipped with the same kind of plug as the Boe-Bot battery pack (2.1 mm, center-positive).



Select Vin if you are using the battery pack that comes with the Boe-Bot kits.

Select Vdd if you are using a DC supply that plugs into an AC outlet (AC adapter).

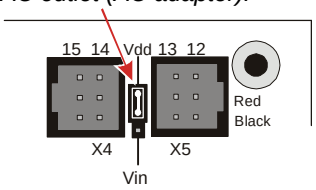


Figure 2-12
Selecting Your Servo Ports' Power Supply on the Board of Education

All examples and instructions in this book will use the battery pack. Figure 2-13 shows the schematic of the circuit you will build on the Board of Education. The jumper is set to Vin.

✓ Connect your servos to your Board of Education as shown in Figure 2-13.

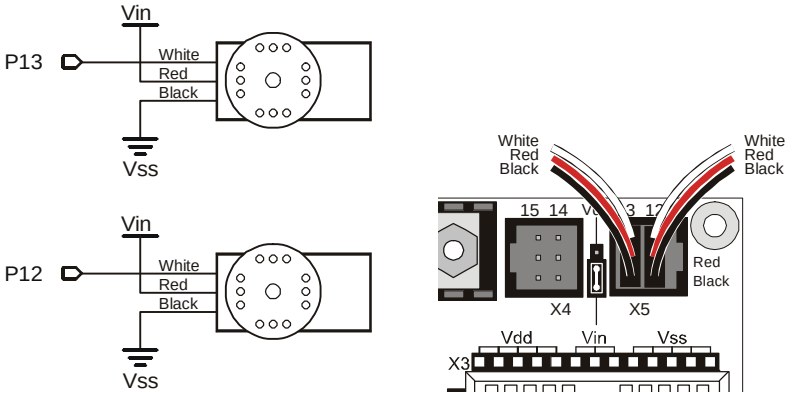


Figure 2-13
Servo Connections for the Board of Education

How do I tell which servo is connected to P13 and which servo is connected to P12? You just plugged your servos into headers with numbers above them. If the number above the header where the servo is plugged in is 13, it means the servo is connected to P13. If the number is 12, it means it's connected to P12.

✓ When you are done assembling the system, it should resemble Figure 2-14 (LED circuits not shown).

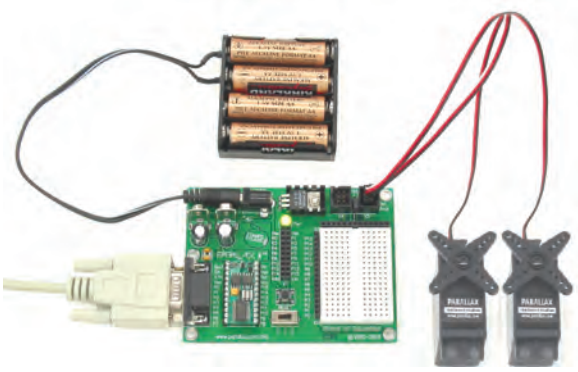


Figure 2-14
Board of Education with Servos and Battery Pack Connected

✓ If you removed the LED circuits after Activity #2, make sure to rebuild them as shown in Figure 2-15. They will be your servo signal monitoring circuits.

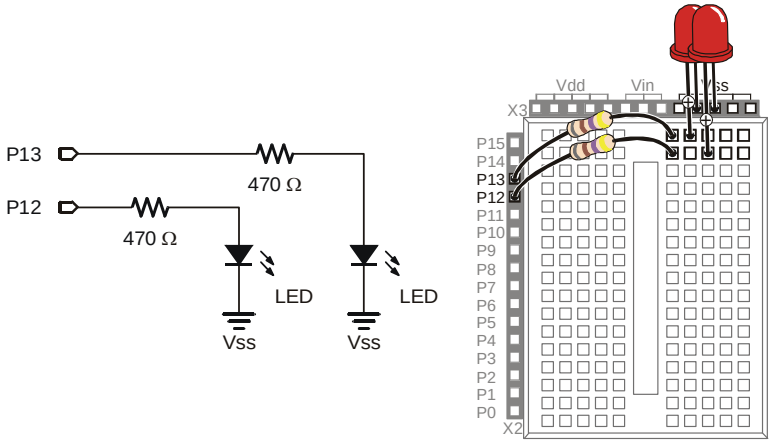


Figure 2-15
LED Servo Signal Monitor Circuit

Disconnecting Power for the Board of Education

Never leave the power connected to your system when you are not working on it.

✓ To disconnect power from your Board of Education, move the 3-position switch to position-0.

✓ Move on to Activity #4: Centering the Servos on page 49.

Connecting the Servos to the BASIC Stamp HomeWork Board

If you are connecting your servos to a BASIC Stamp HomeWork Board, you will need the parts listed below and shown in Figure 2-16:

Parts List:

(1) Battery pack with tinned leads (not included, see Appendix A)
(2) Parallax Continuous Rotation Servos
(2) 3-pin male-male headers (not included, see Appendix A)
(4) Jumper wires
(4) AA batteries – 1.5 V alkaline
(2) Built and tested LED circuits from the previous activity

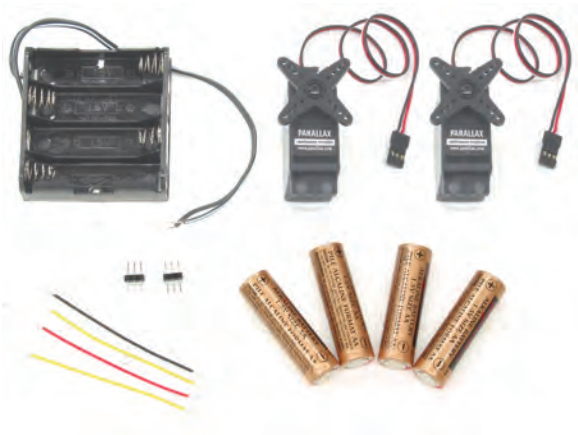


Figure 2-16
Servo Centering Parts for the HomeWork Board

Figure 2-17 shows a schematic of the servo circuits on the HomeWork Board. Before you start building this circuit, make sure that power is disconnected from the BASIC Stamp HomeWork Board.

- ✓ The 9 V battery should be disconnected from the battery clip, and the battery pack with tinned leads should not have any batteries loaded.

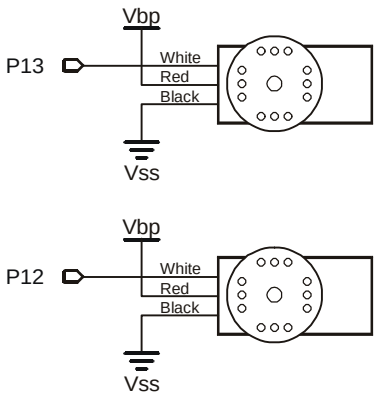


Figure 2-17
Servo Connection Schematic for the BASIC Stamp HomeWork Board

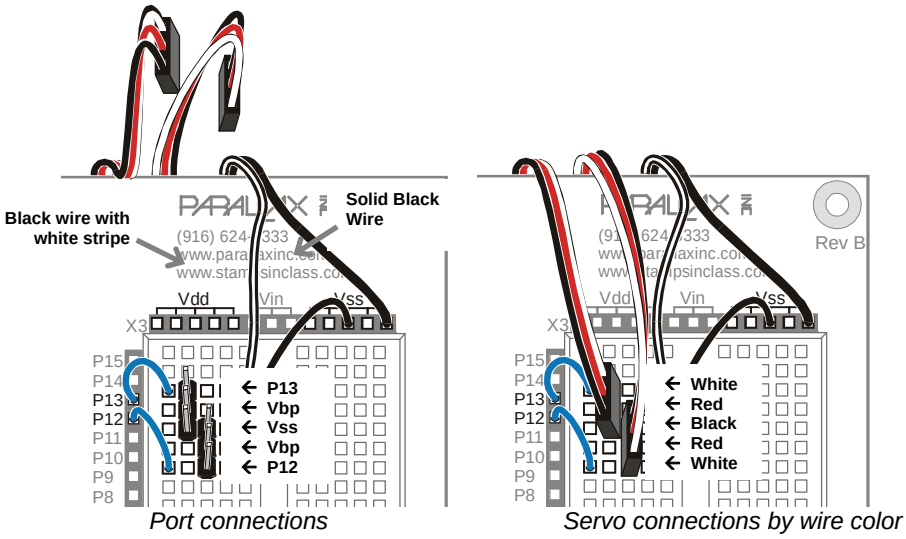
Note: Vbp stands for Voltage Battery Pack. See the i-box below.

- ✓ Remove the two LED/resistor circuits, and save the parts.
- ✓ Build the servo ports shown on the left side of Figure 2-18.
- ✓ Double-check to make sure the black wire with the white stripe is connected to Vbp, and the solid black wire should be connected to Vss.
- ✓ Double-check to make sure that all the connections for P13, Vbp, Vss, Vbp (another one), and P12 all exactly match the wiring diagram.
- ✓ Connect the servo plugs to the male headers as shown in Figure 2-18, on the right side of the figure.
- ✓ Double-check to make sure the servo wire colors match the legend in the figure.

Vbp stands for Voltage battery pack. It refers to the 6 VDC supplied by the four 1.5 V batteries. This is brought directly to the breadboard to power the servos for Boe-Bots built with the HomeWork Board. Your BASIC Stamp is still powered by the 9 V battery.

About Rechargeable Batteries. The Boe-Bot requires 6 V, easily obtained from 4 AA 1.5 V batteries. Alkaline AA batteries are 1.5 V. However, many rechargeable AA batteries supply only 1.2 V, giving a total of 4.8 V, which is not enough to power the BASIC Stamp and Boe-Bot. If you cannot find 1.5 V rechargeable batteries, you may use the inexpensive Boe-Boost (#30078) to add a 5th 1.2 V rechargeable battery, bringing the total back to 6 V.

Figure 2-18: Servo Connection Wiring Diagram for the BASIC Stamp HomeWork Board



Port connections *Servo connections by wire color*

Your setup will then resemble Figure 2-19.

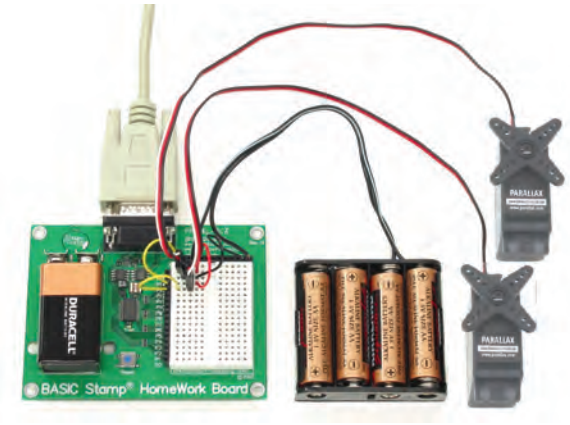


Figure 2-19
Dual Supplies and Servos Connected

✓ Rebuild the LED circuit as shown in Figure 2-20.

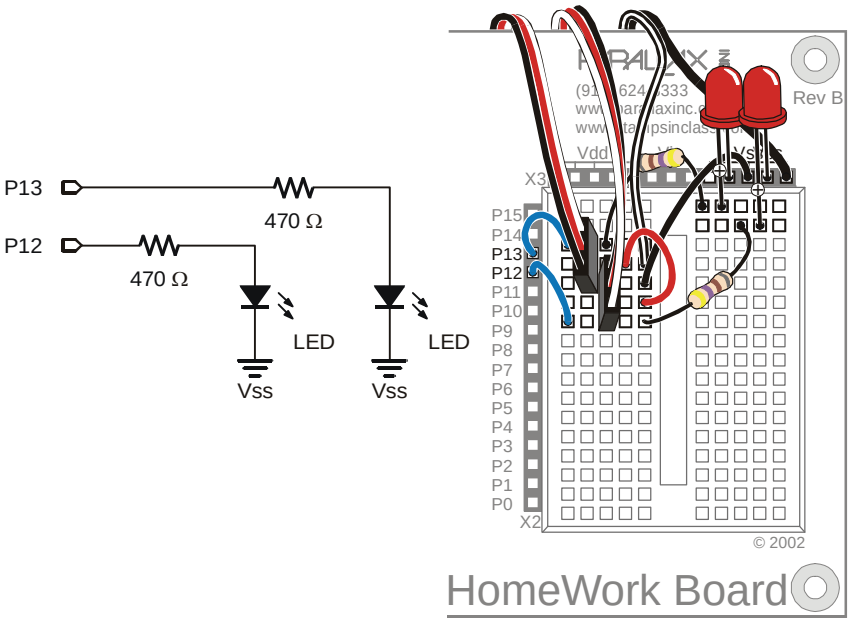


Figure 2-20
LED Servo Signal Monitor Circuit

✓ When all your connections are made and double-checked, load the battery pack with batteries and reconnect the 9 V battery to the HomeWork Board's battery clip.

Disconnecting Power for the HomeWork Board

Never leave the power connected to your system when you are not working with it. From here onward, disconnecting power takes two steps:

- ✓ Unplug the 9 V battery from the battery clip to disconnect power from the HomeWork Board. This disconnects power from the embedded BASIC Stamp, and the power sockets above the breadboard (Vdd, Vin, and Vss).
- ✓ Remove one battery from the battery pack. This disconnects power from the servos.

✓ Move on to Activity #4: Centering the Servos.

ACTIVITY #4: CENTERING THE SERVOS

In this activity, you will run a program that sends the servos a signal, instructing them to stay still. Because the servos are not pre-adjusted at the factory, they will instead start turning. You will then use a screwdriver to adjust them so that they stay still. This is called centering the servos. After the adjustment, you will test the servos to make sure they are functioning properly. The test programs will send signals that make the servos turn clockwise and counterclockwise at various speeds.

Servo Tools and Parts

The Parallax screwdriver shown in Figure 2-21 is the only extra tool you will need for this activity. If needed, any Phillips #1 point screwdriver with a 1/8″ (3.18 mm) shaft should do the trick.

Figure 2-21
Parallax Screwdriver

Sending the Center Signal

Figure 2-22 shows the signal that has to be sent to the servo connected to P12 to calibrate it. This is called the center signal, and after the servo has been properly adjusted, this signal instructs it to stay still. The instruction consists of a series of 1.5 ms pulses with 20 ms pauses between each pulse.

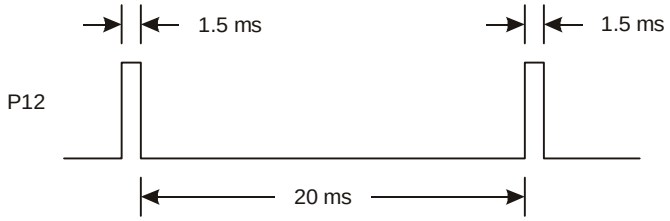


Figure 2-22
Timing Diagram for CenterServoP12.bs2

The 1.5 ms pulses instruct the servo to remain still.

The program for this signal will be a **PULSOUT** command and a **PAUSE** command inside a **DO…LOOP**. Figuring out the **PAUSE** command from the timing diagram is easy; it's going to be **PAUSE 20** for the 20 ms between pulses.

Figuring out the **PULSOUT** command's ***Pin*** argument isn't that hard either; it's going to be 12, for I/O pin P12. Next, let's figure out what the **PULSOUT** command's ***Duration***

argument has to be for 1.5 ms pulses. 1.5 ms is 1.5 thousandths of a second, or 0.0015 s. Remember whatever number is in the **PULSOUT** command's ***Duration*** argument, multiply that number by 2 µs (2 millionths of a second = 0.000002 s), and you will know how long the pulse will last. You can also figure out what the **PULSOUT** command's ***Duration*** argument has to be if you know how long you want the pulse to last. Just divide 2 µs into the time you want the pulse to last. With this calculation:

$$Duration\ argument = \frac{Pulse\ duration}{2\mu s} \frac{0.0015\,s}{0.000002\,s} = 750$$

...we now know that the command for a 1.5 ms pulse to P12 will be **PULSOUT 12, 750**.

It's best to only center one servo at a time, because that way you can hear when the motor stops as you are adjusting it. This program will only send the center signal to the servo connected to P12, and these next instructions will guide you through adjusting it. After you complete the process with the P12 servo, you will repeat it with the servo connected to P13.

- ✓ If you have a Board of Education, make sure to set the 3-position power switch to position-2 as shown in Figure 2-23.

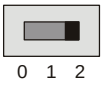


Figure 2-23
Set the 3-Position Switch to Position-2

- ✓ If you are using the HomeWork Board, check the power connections to both your BASIC Stamp and your servos. The 9 V battery should be attached to the battery clip, and the 6 V battery pack should have all four batteries loaded.

If the servos start running (or twitching) as soon as you connect power:

It's probably because the BASIC Stamp is running a program you ran in a previous activity.

- ✓ Make sure to enter, save, and run CenterServoP12.bs2 before continuing to the servo centering instructions that follow the example program.

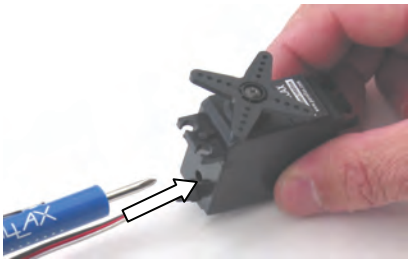
- ✓ Enter, save, and run CenterServoP12.bs2, then continue with the instructions that follow the program.

Example Program: CenterServoP12.bs2

```
' Robotics with the Boe-Bot - CenterServoP12.bs2
' This program sends 1.5 ms pulses to the servo connected to
' P12 for manual centering.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

DO
  PULSOUT 12, 750
  PAUSE 20
LOOP
```

If the servo has not yet been centered, its horn will start turning, and you will be able to hear the motor inside making a whining noise.

- ✓ If the servo is not yet centered, use a screwdriver to gently adjust the potentiometer in the servo as shown in Figure 2-24. Adjust the potentiometer until you find the setting that makes the servo stop turning.



1) Insert tip of Phillips screwdriver into potentiometer access hole.

2) Gently turn screwdriver to adjust potentiometer until the servo stops moving.

Figure 2-24
Center Adjusting a Servo

Caution: do not push too hard with the screwdriver! The potentiometer inside the servo is pretty delicate, so be careful not to apply any more pressure than necessary.

- ✓ Verify that the LED signal monitor circuit connected to P12 is showing activity. It should be emitting light, indicating that the pulses are being transmitted to the servo connected to P12.

If the servo has already been centered, it will not turn. It is unlikely, but a damaged or defective servo would also not turn. Activity #6 will rule out this possibility before the servos are installed on your Boe-Bot chassis.

- ✓ If the servo does not turn, go to the Your Turn section so that you can test and center the other servo that's connected to P13.

What's a Potentiometer? A potentiometer is kind of like an adjustable resistor. The resistance of a potentiometer is adjusted with a moving part. On some potentiometers, this moving part is a knob or a sliding bar, others have sockets that can be adjusted with screwdrivers. The resistance of the potentiometer inside the Parallax Continuous Rotation servo is adjusted with a #1 point Phillips screwdriver tip. You can learn more about potentiometers in *What's a Microcontroller?* and *Basic Analog and Digital* student guides.

Your Turn – Centering the Servo Connected to P13

- ✓ Repeat the process for the servo connected to P13 using this program:

Example Program: CenterServoP13.bs2

```
' Robotics with the Boe-Bot - CenterServoP13.bs2
' This program sends 1.5 ms pulses to the servo connected to
' P13 for manual centering.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

DO
  PULSOUT 13, 750
  PAUSE 20
LOOP
```

Remember to completely disconnect power when you are done.

If you have a Board of Education:

- ✓ Move the 3-position switch to position-0.

If you have a BASIC Stamp HomeWork Board:

- ✓ Unplug the 9 V battery from the battery clip to disconnect power to the HomeWork Board, and:
- ✓ Remove one battery from the battery pack.

ACTIVITY #5: HOW TO STORE VALUES AND COUNT

This activity introduces variables, which are used in PBASIC programs to store values. Boe-Bot programs later in this book will rely heavily on variables. The most important thing about being able to store values is that the program can use them to count. As soon as your program can count, it can both control and keep track of the number of times something happens.

Your servos do not need to be connected to power for this activity.

- ✓ If you have a Board of Education, set the 3-position switch to position-1. The BASIC Stamp, Vdd, Vin, and Vss will all be connected to power, but there will be no power connected to the servo ports
- ✓ If you have a BASIC Stamp HomeWork Board, connect the 9 V battery to the battery clip to power the BASIC Stamp, Vdd, Vin, and Vss. Just leave one battery out of the battery pack to keep power disconnected from the servos.

Using Variables for Storing Values, Math Operations, and Counting

Variables can be used to store values. Before you can use a variable in PBASIC, you have to give it a name and specify its size. This is called declaring a variable.

variableName **VAR** ***Size***

You can declare four different sizes of variables in PBASIC:

Size	**–**	**Stores**
Bit	–	0 to 1
Nib	–	0 to 15
Byte	–	0 to 255
Word	–	0 to 65535, or -32768 to + 32767

The next example program just involves a couple of word variables:

```
value          VAR     Word
anotherValue   VAR     Word
```

After you have declared a variable, you can also initialize it, which means giving it a starting, or initial, value.

```
value = 500
anotherValue = 2000
```

Default Value - If you do not initialize a variable, the program will automatically start by storing the number zero in that variable. That's called the variable's default value.

The "=" sign in **`value = 500`** is an example of an operator. You can use other operators to do math with variables. Here are a couple of multiplication examples:

```
value = 10 * value
anotherValue = 2 * value
```

Example Program: VariablesAndSimpleMath.bs2

This program demonstrates how to declare, initialize, and perform operations on variables.

- ✓ Before running the program, predict what each **`DEBUG`** command will display.
- ✓ Enter, save, and run VariablesAndSimpleMath.bs2.
- ✓ Compare the results to your predictions and explain any differences.

```
' Robotics with the Boe-Bot - VariablesAndSimpleMath.bs2
' Declare variables and use them to solve a few arithmetic problems.
' {$STAMP BS2}
' {$PBASIC 2.5}

value          VAR     Word                  ' Declare variables
anotherValue   VAR     Word

value = 500                                  ' Initialize variables
anotherValue = 2000

DEBUG ? value                                ' Display values
DEBUG ? anotherValue

value = 10 * anotherValue                    ' Perform operations

DEBUG ? value                                ' Display values again
DEBUG ? anotherValue

END
```

How VariablesAndSimpleMath.bs2 Works

This code declares two word variables, **`value`** and **`anotherValue`**.

```
value          VAR     Word      ' Declare variables
anotherValue   VAR     Word
```

These commands are examples of initializing variables to values that you determine. After these two commands are executed, **value** will store 500, and **anotherValue** will store 2000.

```
value = 500                     ' Initialize variables
anotherValue = 2000
```

These **DEBUG** commands help you see what each variable stores after you initialize them. Since value was assigned 500 and **anotherValue** was assigned 2000, these **DEBUG** commands send the messages "value = 500" and "anotherValue = 2000" to the Debug Terminal.

```
DEBUG ? value                   ' Display values
DEBUG ? anotherValue
```

The DEBUG command's "?" formatter can be used before a variable to make the Debug Terminal display its name, the decimal value it's storing, and a carriage return. It's very handy for looking at the contents of a variable.

The riddle in the next three lines is "What will be displayed?" The answer is that **value** will be set equal to ten times **anotherValue**. Since **anotherValue** is 2000, **value** will be set equal to 20,000. The **anotherValue** variable is unchanged.

```
value = 10 * anotherValue       ' Perform operations

DEBUG ? value                   ' Display values again
DEBUG ? anotherValue
```

Your Turn – Calculations with Negative Numbers

If you want to do calculations that involve negative numbers, you can use the **DEBUG** command's **SDEC** formatter to display them. Here's an example that can be made by modifying VariablesAndSimpleMath.bs2.

- ✓ Delete this portion of VariablesAndSimpleMath.bs2:

```
value = 10 * anotherValue       ' Perform operations

DEBUG ? value                   ' Display values again
```

- ✓ Replace it with the following:

```
value = value - anotherValue     ' Answer = -1500

DEBUG "value = ", SDEC value, CR ' Display values again
```

- ✓ Run the modified program and verify that **`value`** changes from 500 to -1500.

Counting and Controlling Repetitions

The most convenient way to control the number of times a piece of code is executed is with a **`FOR…NEXT`** loop. Here is the syntax:

FOR *Counter* = *StartValue* TO *EndValue* {STEP *StepValue*}…NEXT

The three dots "..." indicate that you can put one or more commands between the **`FOR`** and **`NEXT`** statements. Make sure to declare a variable for use in the ***Counter*** argument. The ***StartValue*** and ***EndValue*** arguments can be numbers or variables (or even an expression). When you see something between curly braces { } in a syntax description, it means it's an optional argument. In other words, the **`FOR…NEXT`** loop will work without it, but you can use it for a special purpose.

You don't have to name the variable "counter." For example, you can call it "myCounter."

```
myCounter       VAR     Word
```

Here's an example of a **`FOR…NEXT`** loop that uses the **`myCounter`** variable for counting. It also displays the value of the **`myCounter`** variable each time through the loop.

```
FOR myCounter = 1 TO 10
  DEBUG ? myCounter
  PAUSE 500
NEXT
```

Example Program: CountToTen.bs2

- ✓ Enter, save, and run CountToTen.bs2.

```
' Robotics with the Boe-Bot – CountToTen.bs2
' Use a variable in a FOR...NEXT loop.

' {$STAMP BS2}
' {$PBASIC 2.5}

myCounter          VAR     Word

FOR myCounter = 1 TO 10
  DEBUG ? myCounter
  PAUSE 500
NEXT

DEBUG CR, "All done!"

END
```

Your Turn – Different Start and End Values and Counting in Steps

You can use different values for the ***StartValue*** and ***EndValue*** arguments.

- ✓ Modify the **FOR…NEXT** loop so it looks like this:

```
FOR myCounter = 21 TO 9
  DEBUG ? myCounter
  PAUSE 500
NEXT
```

- ✓ Run the modified program. Did you notice that the BASIC Stamp counted down instead of up? It will do this whenever the ***StartValue*** argument is larger than the ***EndValue*** argument.

Remember the optional {**STEP** ***StepValue***} argument? You can use it to make **myCounter** count in steps. Instead of 9, 10, 11…, you can make it count by twos (9, 11, 13…) or by fives (10, 15, 20…), or whatever ***StepValue*** you give it, forwards or backwards. Here's an example that uses it to count down in steps of 3:

- ✓ Add **STEP 3** to the **FOR…NEXT** loop so it looks like this:

```
FOR myCounter = 21 TO 9 STEP 3
  DEBUG ? myCounter
  PAUSE 500
NEXT
```

- ✓ Run the modified program and verify that it counts backwards in steps of 3.

ACTIVITY #6: TESTING THE SERVOS

There's one last thing to do before assembling your Boe-Bot, and that's testing the servos. In this activity, you will run programs that make the servos turn at different speeds and directions. By doing this, you will verify that your servos are working properly before you assemble your Boe-Bot. This is an example of subsystem testing. Subsystem testing is a worthwhile habit to develop, because it isn't any fun to take a robot back apart just to fix a problem that you could have otherwise caught before putting it together!

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. It's also an essential skill used by engineers worldwide 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

Recall from centering the servos that a signal with a pulse width of 1.5 ms caused the servos to stay still. This was done using a **`PULSOUT`** command with a ***Duration*** of 750. What would happen if the signal's pulse width is not 1.5 ms?

In the Your Turn section of Activity #2, you programmed the BASIC Stamp to send series of 1.3 ms pulses to an LED. Let's take a closer look at that series of pulses and find out how it can be used to control a servo. Figure 2-25 shows how a Parallax Continuous Rotation servo turns full speed clockwise when you send it 1.3 ms pulses. Full speed ranges from 50 to 60 RPM.

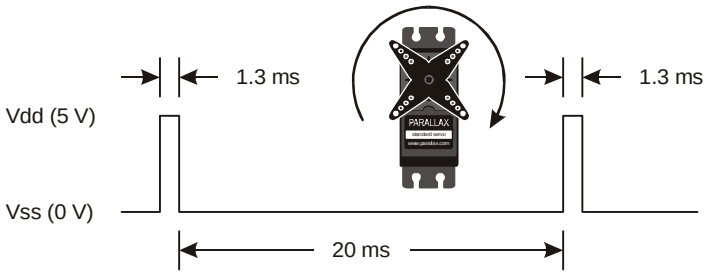


Figure 2-25
A 1.3 ms Pulse Train Turns the Servo Full Speed Clockwise

What's RPM? Revolutions Per Minute. It's the number of full circles something turns in a minute.

What's a pulse train? Just as a railroad train is a series of cars, a pulse train is a series of pulses.

ServoP13Clockwise.bs2 sends this pulse train to the servo connected to P13.

- ✓ Enter, save, and run ServoP13Clockwise.bs2.
- ✓ Verify that the servo's horn is rotating between 50 and 60 RPM clockwise.

```
' Robotics with the Boe-Bot – ServoP13Clockwise.bs2
' Run the servo connected to P13 at full speed clockwise.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

DO
  PULSOUT 13, 650
  PAUSE 20
LOOP
```

Notice that a 1.3 ms pulse requires a **PULSOUT** command ***Duration*** argument of 650, which is less than 750. All pulse widths less than 1.5 ms, and therefore **PULSOUT** ***Duration*** arguments less than 750, will cause the servo to rotate clockwise.

Example Program: ServoP12Clockwise.bs2

By changing the **PULSOUT** command's ***Pin*** argument from 13 to 12, you can make the servo connected to P12 turn full speed clockwise.

- ✓ Save ServoP13Clockwise.bs2 as ServoP12Clockwise.bs2.
- ✓ Modify the program by updating the comments and the **PULSOUT** command's ***Pin*** argument to 12.
- ✓ Run the program and verify that the servo connected to P12 is now rotating between 50 and 60 RPM clockwise.

```
' Robotics with the Boe-Bot – ServoP12Clockwise.bs2
' Run the servo connected to P12 at full speed clockwise.
' {$STAMP BS2}
' {$PBASIC 2.5}
```

```
DEBUG "Program Running!"

DO
  PULSOUT 12, 650
  PAUSE 20
LOOP
```

Example Program: ServoP12Counterclockwise.bs2

You probably guessed that making the **PULSOUT** command's ***Duration*** argument greater than 750 causes the servo to rotate counterclockwise. A ***Duration*** of 850 will send 1.7 ms pulses. This will make the servo turn full speed counterclockwise as shown in Figure 2-26.

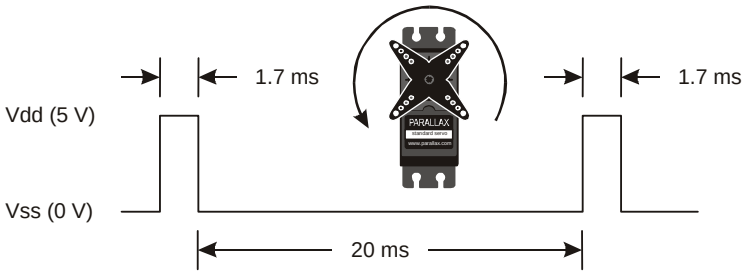


Figure 2-26
A 1.7 ms Pulse Train Makes the Servo Turn Full Speed Counterclockwise

- ✓ Save ServoP12Clockwise.bs2 as ServoP12Counterclockwise.bs2.
- ✓ Modify the program by changing the **PULSOUT** command's ***Duration*** argument from 650 to 850.
- ✓ Run the program and verify that the servo connected to P12 is now rotating between 50 and 60 RPM counterclockwise.

```
' Robotics with the Boe-Bot – ServoP12Counterclockwise.bs2
' Run the servo connected to P12 at full speed counterclockwise.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

DO
  PULSOUT 12, 850
  PAUSE 20
LOOP
```

Pulse Width Modulation. A voltage that spends certain amounts of time in two different states can be considered as a series of resting states and a pulses. Here is a list of different pulse signals that control your servo speed and direction:

- Figure 2-22 on page 49: 1.5 ms high makes the servo hold still.
- Figure 2-25 on page 58: 1.3 ms high makes the servo turn clockwise.
- Figure 2-26 on page 60: 1.7 ms high makes the servo turn counterclockwise.

These signals spend brief amounts of time at high levels (pulses) that are separated by low signals (resting states). A program can adjust the pulse duration, which is the amount of time that the signal is high. This duration is commonly called pulse width because the amount of time the signal is high looks wider or narrower in a timing diagram or on a device like an oscilloscope that plots voltage against time.

Modulation is the process of adjusting a property of a signal that is being transmitted to make it convey certain information. With a servo, the property that is modulated is the pulse width, the amount of time the signal is high. The information it conveys is servo speed and direction.

The servo control signals are examples of positive pulses, with low resting states and high active states. Negative pulses would be the inverted version with high resting states and low active states.

Your Turn – P13Clockwise.bs2

- ✓ Modify the **`PULSOUT`** command's ***Pin*** argument so that it makes the servo connected to P13 turn counterclockwise.

Example Program: ServosP13CcwP12Cw.bs2

You can use two **`PULSOUT`** commands to make both servos turn at the same time. You can also make them turn in opposite directions.

- ✓ Enter, save, and run ServosP13CcwP12Cw.bs2.
- ✓ Verify that the servo connected to P13 is turning full speed counterclockwise while the one connected to P12 is turning full speed clockwise.

```
' Robotics with the Boe-Bot - ServosP13CcwP12Cw.bs2
' Run the servo connected to P13 at full speed counterclockwise
' and the servo connected to P12 at full speed clockwise.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

DO
  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20
LOOP
```

This will be important soon. Think about it: when the servos are mounted on either side of the chassis, one will have to rotate clockwise while the other rotates counterclockwise to make the Boe-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 and re-run the program.

Your Turn – Adjusting the Speed and Direction

There are four different combinations of **PULSOUT** ***Duration*** arguments that will be used repeatedly when programming your Boe-Bot's motion in the upcoming chapters. ServosP13CcwP12Cw.bs2 sends one of these combinations, 850 to P13 and 650 to P12. By testing several possible combinations and filling in the Description column of Table 2-1, you will become familiar with them and build a reference for yourself. You will fill in the Behavior column after your Boe-Bot is fully assembled, when you can see how each combination makes it move.

- ✓ Try the following **PULSOUT** ***Duration*** combinations, and fill in the Description column with your results.

Table 2-1: PULSOUT Duration Combinations			
Durations		**Description**	**Behavior**
P13	**P12**		
850	650	Full speed, P13 servo counter-clockwise, P12 servo clockwise.	
650	850		
850	850		
650	650		
750	850		
650	750		
750	750	Both servos should stay still because of the centering adjustments made in Activity #4.	
760	740		
770	730		
850	700		
800	650		

FOR…NEXT to Control Servo Run Time

Hopefully, by now you fully understand that pulse width controls the speed and direction of a Parallax Continuous Rotation servo. It's a pretty simple way to control motor speed and direction. There is also a simple way to control the amount of time a motor runs, and that's with a **FOR…NEXT** loop.

Here is an example of a **FOR…NEXT** loop that will make the servo turn for a few seconds:

```
FOR counter = 1 TO 100
  PULSOUT 13, 850
  PAUSE 20
NEXT
```

Let's figure out the exact length of time this code would cause the servo to turn. Each time through the loop, the **PULSOUT** command lasts for 1.7 ms, the **PAUSE** command lasts for 20 ms, and it takes around 1.3 ms for the loop to execute.

One time through the loop = 1.7 ms + 20 ms + 1.3 ms = 23.0 ms.

Since the loop executes 100 times, that's 23.0 ms times 100.

$$\begin{aligned} time &= 100 \times 23.0ms \\ &= 100 \times 0.0230s \\ &= 2.30s \end{aligned}$$

Let's say you want the servo to run for 4.6 seconds. Your **FOR…NEXT** loop will have to execute twice as many times:

```
FOR counter = 1 TO 200
  PULSOUT 13, 850
  PAUSE 20
NEXT
```

Example Program: ControlServoRunTimes.bs2

- ✓ Enter, save, and run ControlServoRunTimes.bs2.
- ✓ Verify that the P13 servo turns counterclockwise for about 2.3 seconds, followed by the P12 servo turning for twice as long.

```
' Robotics with the Boe-Bot - ControlServoRunTimes.bs2
' Run the P13 servo at full speed counterclockwise for 2.3 s, then
' run the P12 servo for twice as long.
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

counter VAR Byte

FOR counter = 1 TO 100
  PULSOUT 13, 850
  PAUSE 20
NEXT

FOR counter = 1 TO 200
  PULSOUT 12, 850
  PAUSE 20
NEXT

END
```

Let's say you want to run both servos, the P13 servo at a pulse width of 850 and the P12 servo at a pulse width of 650. Now, each time through the loop, it will take:

1.7ms	*–*	*Servo connected to P13*
1.3 ms	*–*	*Servo connected to P12*
20 ms	*–*	*Pause duration*
1.6 ms	*–*	*Code overhead*
---		---
24.6 ms	*–*	*Total*

If you want to run the servos for a certain amount of time, you can calculate it like this:

Number of pulses = Time s / 0.0246 s = Time / 0.0246

Lets' say we want to run the servos for 3 seconds. That's:

Number of pulses = 3 / 0.0246 = 122

Now, you can use the value 122 in the ***EndValue*** of the **FOR…NEXT** loop, and it will look like this:

```
FOR counter = 1 TO 122
  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20
NEXT
```

Example Program: BothServosThreeSeconds.bs2

Here's an example of making the servos turn in one direction for three seconds, then reversing their direction.

- ✓ Enter, save, and run BothServosThreeSeconds.bs2.

```
' Robotics with the Boe-Bot - BothServosThreeSeconds.bs2
' Run both servos in opposite directions for three seconds, then reverse
' the direction of both servos and run another three seconds.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

counter VAR Byte

FOR counter = 1 TO 122
  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20
NEXT

FOR counter = 1 TO 122
  PULSOUT 13, 650
  PULSOUT 12, 850
  PAUSE 20
NEXT

END
```

- ✓ Verify that each servo turned one direction for three seconds, and then reversed direction and turned for three more seconds. Did you notice that while the servos reversed at the same moment, they were always turning in opposite directions? Why would this be useful?

Your Turn – Predict Servo Run Time

- ✓ Pick a time (six seconds or less), that you want your servos to turn.
- ✓ Divide the number of seconds by 0.024.
- ✓ Your answer is the number of loops you will need.
- ✓ Modify BothServosThreeSeconds.bs2 so that it makes both servos run for the amount of time you selected.
- ✓ Compare your predicted run time to the actual run time.
- ✓ Remember to disconnect power from your system (board and servos) when you are done.

TIP – To measure the run time, press and hold the Reset button on your Board of Education (or BASIC Stamp HomeWork Board). When you are ready to start timing, let go of the Reset button.

SUMMARY

This chapter guided you through connecting, adjusting, and testing the Parallax Continuous Rotation servos. Along the way, a variety of PBASIC commands were introduced. The **PAUSE** command makes the program stop for brief or long periods of time, depending on the ***Duration*** argument you use. **DO…LOOP** makes repeating a single or group of PBASIC commands over and over again efficient. **HIGH** and **LOW** were introduced as a way of making the BASIC Stamp connect an I/O pin to Vdd or Vss. High and low signals were viewed with the help of an LED circuit. These signals were used to introduce timing diagrams.

The **PULSOUT** command was introduced as a more precise way to deliver a high or low signal, and an LED circuit was also used to view signals sent by the **PULSOUT** command. **DO…LOOP**, **PULSOUT**, and **PAUSE** were then used to send the Parallax Continuous Rotation servos the signal to stay still, which is 1.5 ms pulses every 20 ms. The servo was adjusted with a screwdriver while receiving the 1.5 ms pulses until it stayed still. This process is called "centering" the servo.

After the servos were centered, variables were introduced as a way to store values. Variables can be used in math operations and counting. **FOR…NEXT** loops were introduced as a way to count. **FOR…NEXT** loops control the number of times the lines of code between the **FOR** and **NEXT** statements are executed. **FOR…NEXT** loops were then used to control the number of pulses delivered to a servo, which in turn controls the amount of time the servo runs.

Questions

1. How do the Parallax Continuous Rotation servos differ from standard servos?
2. How long does a millisecond last? How do you abbreviate it?
3. What PBASIC commands can you use to make other PBASIC commands execute over and over again?
4. What command causes the BASIC Stamp to internally connect one of its I/O pins to Vdd? What command makes the same kind of connection, but to Vss?
5. What are the names of the different size variables that can be declared in a PBASIC program? What size values can each size of variable store?
6. What is the key to controlling a Parallax Continuous Rotation servo's speed and direction? How does this relate to timing diagrams? How does it relate to PBASIC commands? What the command and argument can you adjust to control a continuous rotation servo's speed and direction?

Exercises

1. Write a **PAUSE** command that makes the BASIC Stamp do nothing for 10 seconds.
2. Modify this **FOR…NEXT** loop so that it counts from 6 to 24 in steps of 3. Also, write the variable declaration you will need to make this program work.

```
FOR counter = 9 TO 21
  DEBUG ? counter
  PAUSE 500
NEXT
```

Project

1. Write a program that causes an LED connected to P14 to light dimly (on/off with every pulse) while the P12 servo is turning.
2. Write a program that takes the servos through three seconds of each of the four different combinations of rotation. Hint: you will need four different **FOR…NEXT** loops. First, both servos should rotate counterclockwise, then they should both rotate clockwise. Then, the P12 servo should rotate clockwise as the P13 servo rotates counterclockwise, and finally, the P12 servo should rotate counterclockwise while the P13 servo rotates clockwise.

Solutions

Q1. Instead of holding a certain position like a standard servo, the Parallax Continuous Rotation servos turn a certain direction at a certain speed.
Q2. A millisecond lasts one thousandth of a second, and "ms" is the abbreviation.
Q3. The **DO…LOOP** command is used to make other PBASIC commands execute over and over.
Q4. **HIGH** connects I/O pin to Vdd, **LOW** connects I/O pin to Vss.
Q5. The variable sizes are bit, nib, byte, and word.

Bit – Stores 0 to 1
Nib – Stores 0 to 15
Byte – Stores 0 to 255
Word – Stores 0 to 65535 or -32768 to +32767

Q6. Pulse width controls servo speed and direction. As seen on a timing diagram, the pulse width is the high time. In PBASIC, the pulse can be generated with the **PULSOUT** command. The **PULSOUT** command's ***Duration*** argument adjusts the speed and direction.

E1. **PAUSE 10000**
E2. The key to writing the variable declaration is to choose a variable size large enough to hold the value 24. A **Nib** will not work, since the maximum value a nibble can store is 15. Therefore, choose a Byte variable.

```
counter VAR Byte
FOR counter = 6 TO 24 STEP 3
  DEBUG ? counter
  PAUSE 500
NEXT
```

P1. The key to solving this problem is to send a pulse train to the LED as well as the servo.

```
' Robotics with the Boe-Bot - Ch02Prj01_DimlyLitLED.bs2
' Run servo and send same signal to dimly light the LED on P14.
'{$STAMP BS2}
'{$PBASIC 2.5}

DEBUG "Program Running!"

DO
  PULSOUT 12, 650                          ' P12 servo clockwise
  PULSOUT 14, 650                          ' P14 LED lights dimly
  PAUSE 20
LOOP
```

P2. First, calculate the number of loops needed to get the servos to run for three seconds, for each combination of rotation. As given on page 65, the code overhead is 1.6 ms.

Four combinations (1,2,3,4).
Each combination: Determine **PULSOUT** ***Duration*** arguments:

1. Both counterclockwise: 12, 850 and 13, 850
2. Both clockwise: 12, 650 and 13, 650
3. 12 CW and 13 CCW: 12, 850 and 13, 650
4. 12 CCW and 13 CW: 12, 650 and 13, 850

Each combination: Calculate how long it will take for one loop:

1. one loop = 1.7 + 1.7 + 20 ms + 1.6 = 25.0 ms = 0.025 s
2. one loop = 1.3 + 1.3 + 20 ms + 1.6 = 24.2 ms = 0.0242 s
3. one loop = 1.7 + 1.3 + 20 ms + 1.6 = 24.6 ms = 0.0246 s
4. one loop = 1.3 + 1.7 + 20 ms + 1.6 = 24.6 ms = 0.0246 s

Each combination: Calculate number of pulses needed for 3 s of running:

1. number of pulses = 3 s / 0.025 s = 120
2. number of pulses = 3 s / 0.0242 s = 123.9 = 124
3. number of pulses = 3 s / 0.0246 s = 121.9 = 122
4. number of pulses = 3 s / 0.0246 s = 121.9 = 122

Now write four **FOR…NEXT** loops, using the number of pulses calculated for the ***EndValue*** argument. Include the correct **PULSOUT** arguments for the combination of rotation.

```
' Robotics with the Boe-Bot - Ch02Prj02_4RotationCombinations.bs2
' Move servos through 4 clockwise/counterclockwise rotation
' combinations.

'{$STAMP BS2}
'{$PBASIC 2.5}

DEBUG "Program Running!"

counter        VAR     Word

FOR counter = 1 TO 120             ' Loop for three seconds
  PULSOUT 13, 850                  ' P13 servo counterclockwise
  PULSOUT 12, 850                  ' P12 servo counterclockwise
  PAUSE 20
NEXT

FOR counter = 1 TO 124             ' Loop for three seconds
  PULSOUT 13, 650                  ' P13 servo clockwise
  PULSOUT 12, 650                  ' P12 servo clockwise
  PAUSE 20
NEXT

FOR counter = 1 TO 122             ' Loop for three seconds
  PULSOUT 13, 650                  ' P13 servo clockwise
  PULSOUT 12, 850                  ' P12 servo counterclockwise
  PAUSE 20
NEXT

FOR counter = 1 TO 122             ' Loop for three seconds
  PULSOUT 13, 850                  ' P13 servo counterclockwise
  PULSOUT 12, 650                  ' P12 servo clockwise
  PAUSE 20
NEXT

END
```

Chapter 3: Assemble and Test Your Boe-Bot

This chapter contains instructions for building and testing your Boe-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 lead to mystifying Boe-Bot behavior in later chapters. Here is a summary of what you will do in each of the activities in this chapter:

Activity	Summary
1	Build the Boe-Bot.
2	Re-test the servos to make sure they are properly connected .
3	Connect and test a speaker that can let you know when the Boe-Bot's batteries are low.
4	Use the Debug Terminal to control and test servo speed.

ACTIVITY #1: ASSEMBLING THE BOE-BOT ROBOT

This activity will guide you through assembling the Boe-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.

Servo Tools and Parts

All of the tools shown in Figure 3-1 are common and can be found in most households and school shops. They can also be purchased at local hardware stores.

Tools

(1) Parallax screwdriver
(Phillips #1 point, included)
(1) 1/4″ Combination wrench
(optional but handy)
(1) Needle-nose pliers (optional)

Figure 3-1
Boe-Bot Assembly Tools

Mounting the Topside Hardware

- ✓ Start by gathering this list of parts.
- ✓ Then, follow the accompanying instructions.

Parts List:

See Figure 3-2.

(1) Boe-Bot chassis
(4) 1″ Standoffs
(4) Pan head screws, 1/4″ 4-40
(1) Rubber grommet, 13/32″

Instructions:

- ✓ Insert the 13/32″ rubber grommet into the hole in the center of the Boe-Bot chassis.
- ✓ Make sure the groove in the outer edge of the rubber grommet is seated on the edge of the hole in the chassis.
- ✓ Use the four 1/4″ 4-40 screws to attach the four standoffs to the chassis as shown.

Boe-Bot Parts - The parts for the Boe-Bot are either included in the Boe-Bot full kit or in a combination of the Board of Education Full Kit and Robotics Parts Kit. If you are using a HomeWork Board, you need a battery pack with tinned leads instead of a barrel plug, and two additional 3-pin headers. See Appendix A: Parts List and Kit Options on page 289 for more information.

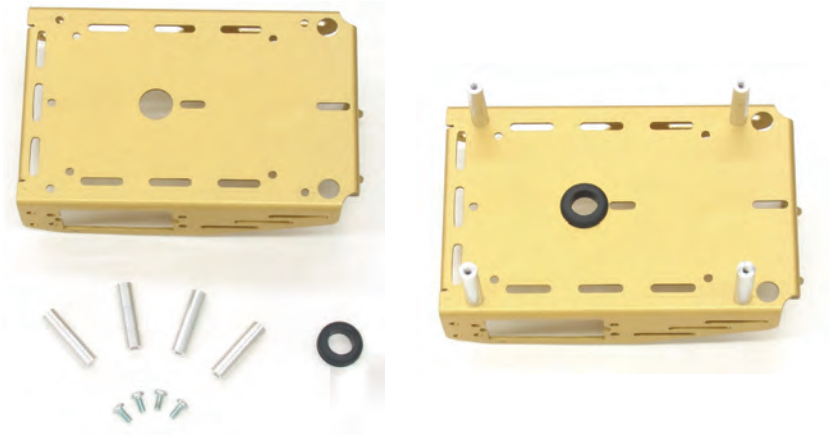


Figure 3-2
Chassis and Topside Hardware

Parts (left); assembled (right)

Removing the Servo Horns

- ✓ Disconnect the power from your BASIC Stamp and servos.
- ✓ Remove all of the AA batteries from the battery pack.
- ✓ Disconnect the servos from your board.

Parts List:

See Figure 3-3.

(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; they will be used in a later step.

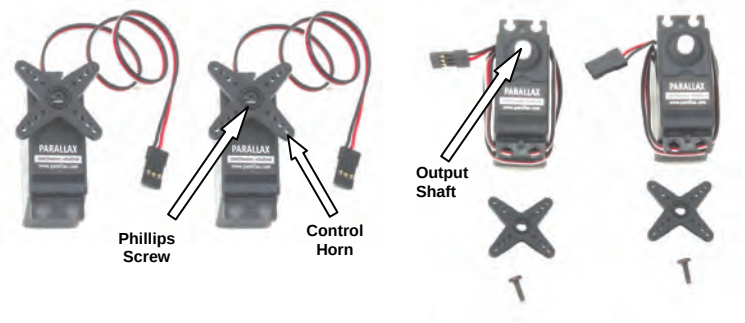


Figure 3-3
Chassis and Topside Hardware

Parts (left); assembled (right)

Stop!

- ✓ Before this next step, you must have completed these activities from Chapter 2: Your Boe-Bot's Servo Motors
  - Activity #3: Connecting the Servo Motors; page 40.
  - Activity #4: Centering the Servos; page 49.

Mounting the Servos on the Chassis

Servo Mounting Options - Agile Maneuvers vs. Potentiometer & Maintenance Access

The photographs in this text show the servos mounted from the inside, and oriented so the potentiometer access port is facing the center of the chassis. This positions the axles close to the center of the Boe-Bot, allowing for agile maneuvering. If you are diligent about centering your servos before building your Boe-Bot, this causes no problems.

Many educators prefer the option of mounting the servos from the outside, and oriented so the potentiometer access port faces the front of the Boe-Bot. This has the advantage of allowing easy access to adjust these potentiometers on an assembled robot, and also for quick replacement of damaged servos. However, the Boe-Bot will have a longer, wider wheel base and be a little less nimble on maneuvers. You may find that you need to adjust some values in your programs slightly to achieve the same results. The choice is yours.

Parts List:

See Figure 3-4.

(2) Boe-Bot Chassis (partially assembled.
(2) Parallax Continuous Rotation servos
(8) Pan Head Screws, 3/8″ 4-40
(8) Nuts, 4-40

Instructions:

- ✓ 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).

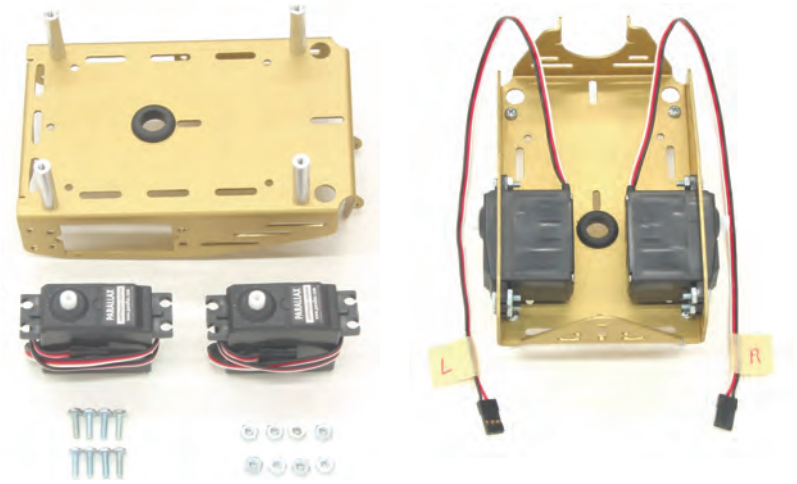


Figure 3-4
Mounting the Servos on the Chassis

Parts (left); assembled (right)

Mounting the Battery Pack

Figure 3-5 shows two different sets of parts. Use the parts on the left if you have a Board of Education, and the parts on the right if you have a HomeWork Board.

Parts List for Boe-Bot with a Board of Education:

See Figure 3-5 (left side).

(1) Boe-Bot chassis (partially assembled)
(2) Flat head Phillips screws, 3/8″ 4-40
(2) Nuts, 4-40
(1) Battery pack with center-positive plug

Parts List for Boe-Bot with a HomeWork Board:

See Figure 3-5 (right side).

(1) Boe-Bot chassis (partially assembled)
(2) Flat head Phillips screws, 3/8″ 4-40
(2) Nuts, 4-40
(1) Battery pack with tinned leads

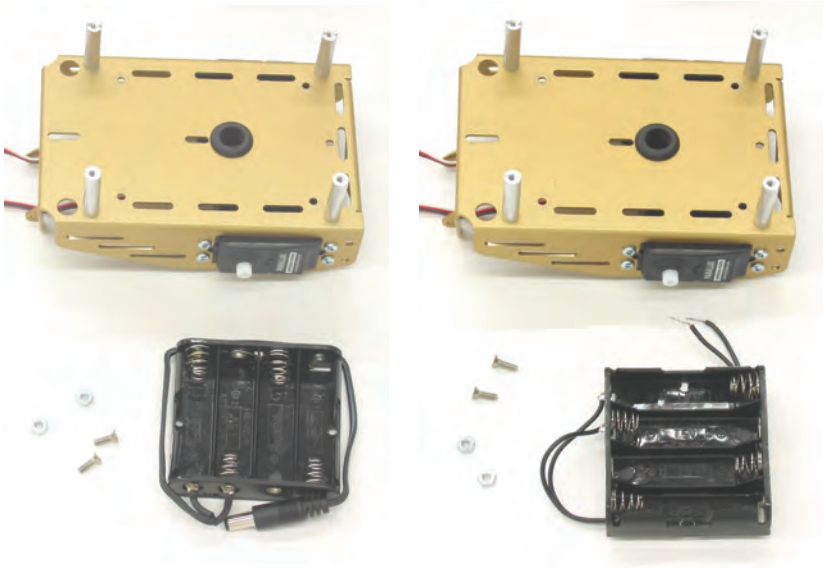


Figure 3-5
Battery Pack Mounting Hardware

Instructions:

- ✓ Use the flathead screws and nuts to attach the battery pack to underside of the Boe-Bot chassis as shown on the left side of Figure 3-6.
- ✓ Make sure to insert the screws through the battery pack, and then tighten down the nuts on the topside of the chassis.

- ✓ As shown on the right side of Figure 3-6, pull the battery pack's power cord through the hole with the rubber grommet in the center of the chassis.
- ✓ Pull the servo lines through the same hole.
- ✓ Arrange the servo lines and supply cable as shown.

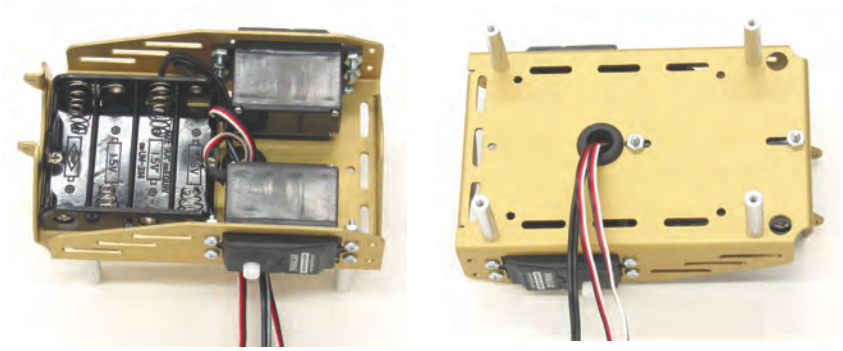


Figure 3-6
Battery Pack Installed

Mounting the Wheels

Parts List:

(1) Partially assembled Boe-Bot (not shown)
(1) 1/16″ Cotter pin
(1) Tail wheel ball
(2) Rubber band tires
(2) Plastic machined wheels
(2) Screws that were saved in the Removing the Servo Horns step

Figure 3-7
Wheel Hardware

Instructions:

The left side of Figure 3-8 shows the Boe-Bot's tail wheel mounted on the chassis. The 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 the hole in the tail wheel up 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.

The right side of Figure 3-8 shows the Boe-Bot's drive wheels mounted on the servos.

- ✓ Stretch each rubber band tire and seat it on the outer edge of each wheel.
- ✓ Each plastic wheel has a recess that fits on a servo output shaft. Press each plastic wheel onto a servo output shaft making sure the shaft lines up with and sinks into the recess.
- ✓ Use the machine screws that you saved when you removed the servo horns to attach the wheels to the servo output shafts.

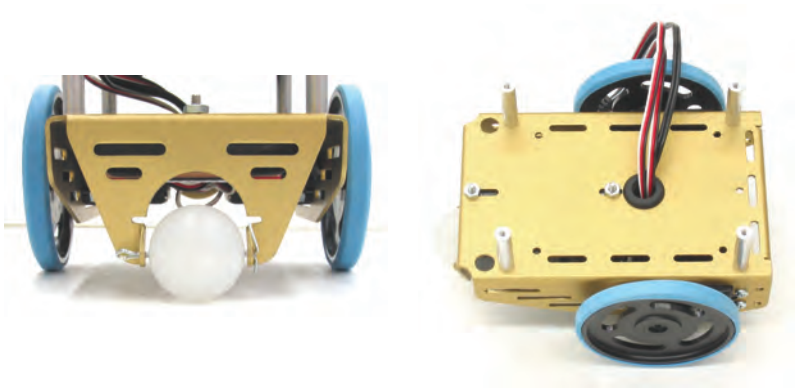


Figure 3-8
Mounting the Wheels

Tail wheel (left); drive wheels (right)

Attaching the Board to the Chassis

Parts List for Boe-Bot with a Board of Education:

See left side of Figure 3-9.

(1) Boe-Bot chassis (partially assembled)
(4) Pan head screws, 1/4″ 4-40
(1) Board of Education with BASIC Stamp 2

Parts List for Boe-Bot with a HomeWork Board:

See right side of Figure 3-9.

(1) Boe-Bot chassis (partially assembled)
(4) Pan head screws, 1/4″ 4-40
(1) BASIC Stamp HomeWork Board

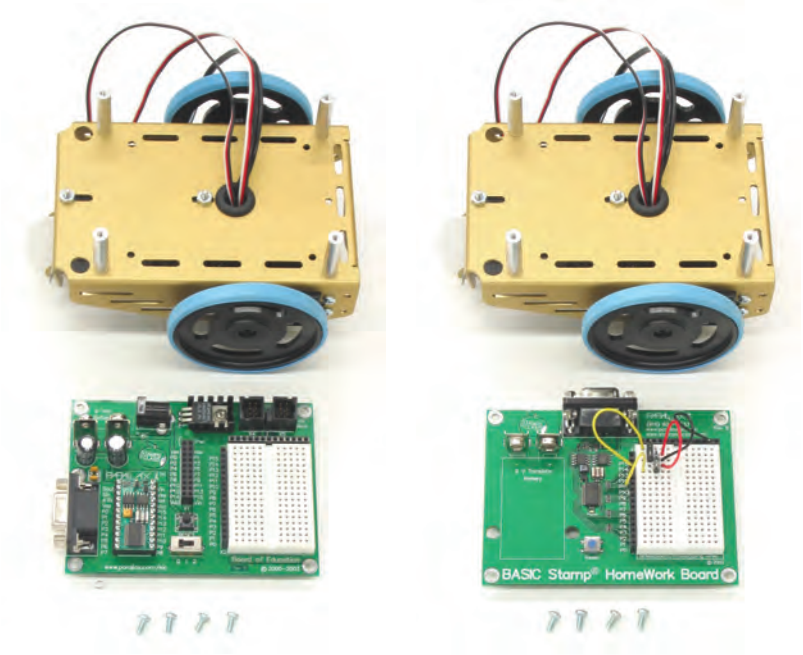
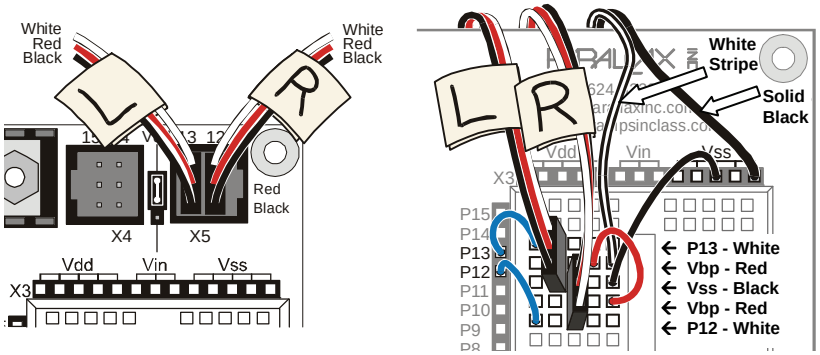


Figure 3-9
Boe-Bot Chassis and Boards

Board of Education (left); HomeWork Board (right)

Figure 3-10 shows the servo ports reconnected for both the Board of Education (left side) and the HomeWork Board (right side).

- ✓ Reconnect the servos to the servo headers.
- ✓ Make sure to connect the plug labeled ‘L’ to the P13 port and the plug labeled ‘R’ to the P12 port.

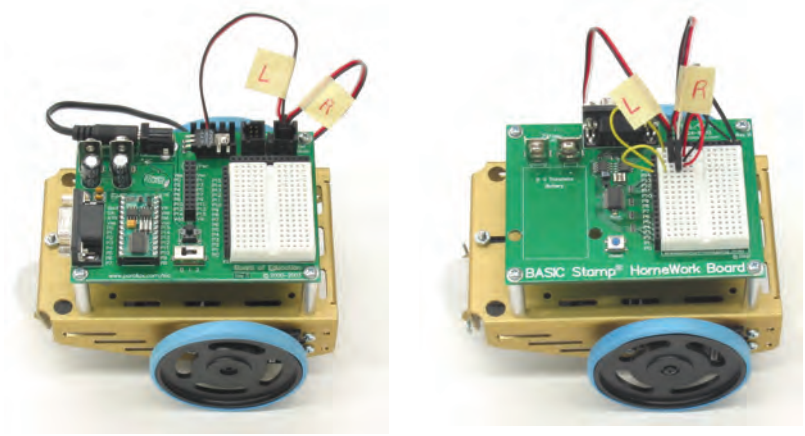


On Board of Education *On HomeWork Board*

Figure 3-10
Servo Ports Reconnected

Figure 3-11 shows the Boe-Bot chassis with their respective boards attached.

- ✓ Set the board on the four standoffs so that they line up with the four 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.

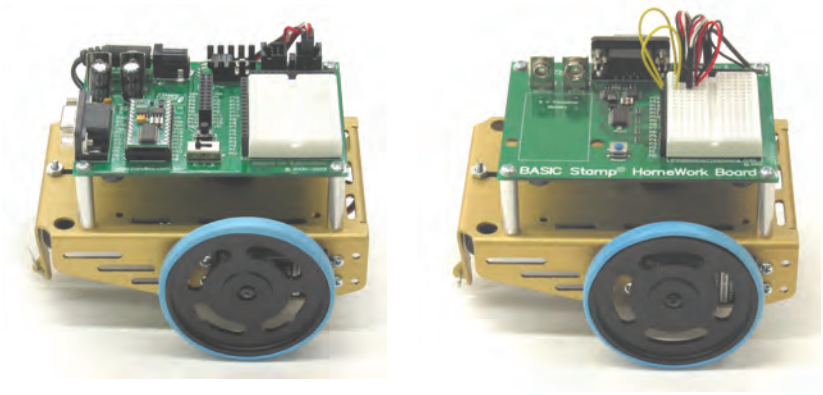


With Board of Education *With HomeWork Board*

Figure 3-11
Boards Attached to Boe-Bot Chassis

Figure 3-12 shows assembled Boe-Bot robots, the left built with a Board of Education (Serial Rev C) and the right built with a HomeWork Board.

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



With Board of Education *With HomeWork Board*

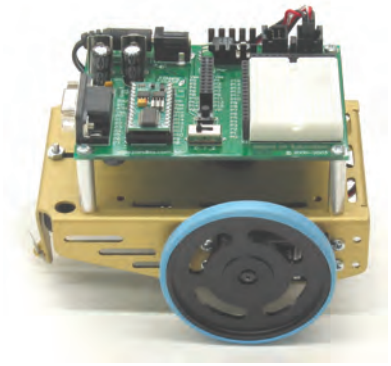
Figure 3-12
Assembled Boe-Bot Robots

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. Figure 3-13 shows your Boe-Bot's front, back, left, and right. We need to make sure that the servo on the right turns when it receives pulses from P12 and that the servo on the left turns when it receives pulses from P13.

Left

Back



Front

Right

Figure 3-13
Your Boe-Bot robot's Front, Back, Left, and Right

Testing the Right Wheel

The next example program will test the servo connected to the right wheel, shown in Figure 3-14. The program will make this wheel turn clockwise for three seconds, then stop for one second, then turn counterclockwise for three seconds.

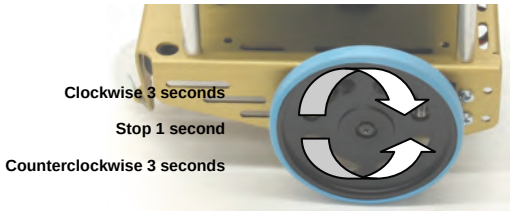


Figure 3-14
Testing the Right Wheel

Example Program: RightServoTest.bs2

- ✓ Set the Boe-Bot on its nose so that the drive wheels are suspended above ground.
- ✓ Reload the batteries into the battery pack.
- ✓ If you have a Board of Education, set the 3-position switch to position-2.
- ✓ If you have a BASIC Stamp HomeWork Board, connect the 9 V battery to the battery clip.
- ✓ Enter, save, and run RightServoTest.bs2.
- ✓ 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 the Servo Troubleshooting section. It comes right after RightServoTest.bs2.
- ✓ If the right wheel/servo does behave properly, then move on to the Your Turn section, where you will test the left wheel.

```
' Robotics with the Boe-Bot - RightServoTest.bs2
' Right servo turns clockwise three seconds, stops 1 second, then
' counterclockwise three seconds.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

counter        VAR     Word

FOR counter = 1 TO 122                      ' Clockwise just under 3 seconds.
  PULSOUT 12, 650
  PAUSE 20
NEXT
```

```
FOR counter = 1 TO 40                          ' Stop one second.
  PULSOUT 12, 750
  PAUSE 20
NEXT

FOR counter = 1 TO 122                         ' Counterclockwise three seconds.
  PULSOUT 12, 850
  PAUSE 20
NEXT

END
```

Your Turn – Testing the Left Wheel

Now, it's time to run the same test on the left wheel as shown in Figure 3-15. This involves modifying RightServoTest.bs2 so that the **PULSOUT** commands are sent to the servo connected to P13 instead of the servo connected to P12.

All you have to do is change the three **PULSOUT** commands so that they read **PULSOUT 13** instead of **PULSOUT 12**.

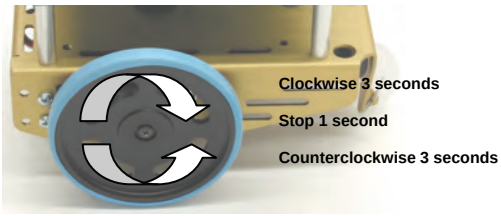


Figure 3-15
Testing the Left Wheel

- ✓ Save RightServoTest.bs2 as LeftServoTest.bs2.
- ✓ Change the three **PULSOUT** commands so that they read **PULSOUT 13** instead of **PULSOUT 12**.
- ✓ Save and then run the program.
- ✓ Verify that it makes the left servo turn clockwise for 3 seconds, stops for 1 second, then makes the servo turn counterclockwise for 3 seconds.
- ✓ If the left wheel/servo does not behave as predicted, see the Servo Troubleshooting box below.
- ✓ If the left wheel/servo does behave properly, then your Boe-Bot is functioning properly, and you are ready to move on to the next activity.

Servo Troubleshooting: Here is a list of some common symptoms and how to fix them.

The servo doesn't turn at all.

- ✓ If you are using a Board of Education, make sure the 3-position switch is set to position-2. You can then re-run the program by pressing and releasing the Reset button.
- ✓ If you are using a BASIC Stamp HomeWork Board, make sure the battery pack has fresh batteries, all oriented properly in the case.
- ✓ Double-check your servo connections Figure 3-10 on page 81 as a guide. If you are using a HomeWork Board, you may also want to take a second look at Figure 2-18 on page 47.
- ✓ Check and make sure you entered the program correctly.

The right servo doesn't turn, but the left one does.

This means that the servos are swapped. The servo that's connected to P12 should be connected to P13, and the servo that's connected to P13 should be connected to P12.

- ✓ Disconnect power.
- ✓ Unplug both servos.
- ✓ Connect the servo that was connected to P12 to P13.
- ✓ Connect the other servo (that was connected to P13) to P12.
- ✓ Reconnect power.
- ✓ Re-run RightServoTest.bs2.

The wheel does not fully stop; it turns slowly.

This means that the servo may not be exactly centered. There are two ways to fix this:

- ✓ Adjusting in hardware: Go back and re-do the Chapter 2, Activity #4: Centering the Servos on page 49. If the servos are not mounted to give easy access to the potentiometer ports, consider re-orienting them for re-assembly.
- ✓ Adjusting in software: If the wheel turns slowly counterclockwise, use a value that's a little smaller than 750. If it's turning clockwise, use a value that's a little larger than 750. This new value will be used in place of 750 for all **PULSOUT** commands for that wheel as you do the experiments in this book.

The wheel doesn't stop for one second between the clockwise and counterclockwise rotations.

The wheel might turn rapidly for three seconds in one direction and four in the other. It might also turn rapidly for three seconds, then just a little slower for one second, then turn rapidly again for three seconds. Or, it might turn rapidly in the same direction for seven seconds. Regardless, it means the potentiometer is out of adjustment.

- ✓ Remove the wheels, un-mount the servos and repeat the exercise in Activity #4: Centering the Servos on page 49.

ACTIVITY #3: START/RESET INDICATOR CIRCUIT AND PROGRAM

When the voltage supply drops below the level a device needs to function properly, it's called brownout. The BASIC Stamp protects itself from brownout by making its processor and program memory chips go dormant until the power supply voltage returns to normal levels. A drop below 5.2 V at Vin results in a drop below 4.3 V at the BASIC Stamp's internal voltage regulator output. A circuit called a brownout detector on the BASIC Stamp is always on the lookout for this condition. When brownout occurs, the brownout detector disables the BASIC Stamp's processor and program memory.

When the supply voltage comes back above 5.2 V, the BASIC Stamp starts running again, but not at the same place in the program. Instead, it starts from the beginning of the program. This is actually the same thing that happens when you unplug power and plug it back in, and it's also the same thing that happens if you press and release the Reset button on your board.

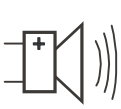
When the Boe-Bot's batteries are running low, brownouts can cause the program to restart when you're not expecting it to. This can lead to some really mystifying Boe-Bot behavior. In some cases, the Boe-Bot will be running whatever course it's programmed to navigate, and all of a sudden, it might seem to get lost and go in an unexpected direction. If low batteries are the cause, it could be the fact that the Boe-Bot's program went back to the beginning and started over again. In other cases, the Boe-Bot can end up doing a confused dance because every time the servos start turning, it overtaxes the already low batteries. The program attempts to make the servos turn for a split second, then restarts, over and over again.

These situations make a program start/reset indicator an extremely useful diagnostic device as well as a useful robot tool. One way to indicate resets is to include an unmistakable signal at the beginning of all the Boe-Bot's programs. The signal occurs every time the power gets plugged in, but it also occurs every time a reset due to brownout conditions occurs. One effective signal for resets is a speaker that emits a tone each time the BASIC Stamp program runs from the beginning or resets.

BASIC Stamp HomeWork Board Special Instructions

Although the reset indicator will tell you when the 9 V battery supplying the BASIC Stamp is running low, it will not tell you when the servo supply (the battery pack) is running low. You can always tell when your battery pack is running low because the servos will gradually move slower and slower during normal operation. When you observe this symptom, replace the dead batteries with new or freshly recharged batteries.

This exercise will introduce a device called a piezoelectric speaker (piezospeaker) that you can use to generate tones. This speaker can make different tones depending on the frequency of high/low signals it receives from the BASIC Stamp. The schematic symbol and part drawing for the piezoelectric speaker are shown in Figure 3-16. This speaker will be used for emitting the tones when the BASIC Stamp is reset in this activity as well as in the rest of the activities in this text.



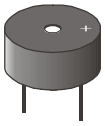


Figure 3-16
Piezospeaker

What's frequency? It's the measurement of how often something occurs in a given amount of time.

What's a piezoelectric element and how can it make sound? It's a crystal that changes shape slightly when voltage is applied to it. By applying high and low voltages to a piezoelectric crystal at a rapid rate, it causes the piezoelectric crystal to rapidly change shape. The result is vibration. Vibrating objects cause the air around them to vibrate also. This is what our ear detects as sounds and tones. Every rate of vibration has a different tone. For example, if you pluck a single guitar string, it will vibrate at one frequency, and you will hear a particular tone. If you pluck a different guitar string, it will vibrate at a different frequency and make a different tone.

Piezoelectric elements have many uses. For example, 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.

Parts Required

(1) Assembled and tested Boe-Bot
(1) Piezospeaker
(misc.) Jumper wires

If your piezospeaker has a label that says "Remove seal after washing" just peel it off and proceed. Your piezospeaker does not need to be washed!

Building the Start/Reset Indicator Circuit

Figure 3-17 shows piezospeaker alarm circuit schematics for both the Board of Education and BASIC Stamp HomeWork Board. Figure 3-18 shows a wiring diagram for each board.

Always disconnect power before building or modifying circuits!

- ✓ If you have a Board of Education, set the 3-position switch to position-0.
- ✓ If you have a BASIC Stamp HomeWork Board, disconnect the 9 V battery from the battery clip and remove a battery from the Battery Pack.

✓ Build the circuit shown in Figure 3-17 and Figure 3-18.

The piezospeaker and servo circuits will remain connected to your board for the rest of the activities in this text.

All circuit schematics from this point onward will show circuits that should be added to the existing servo and piezospeaker circuits.

All wiring diagrams will show the circuit from the schematic that comes just before it along with the servo and piezospeaker circuit connections.

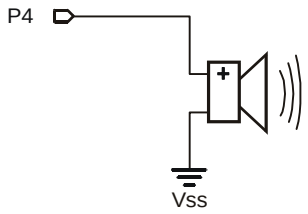


Figure 3-17
Program Start/Reset Indicator Circuit Schematic

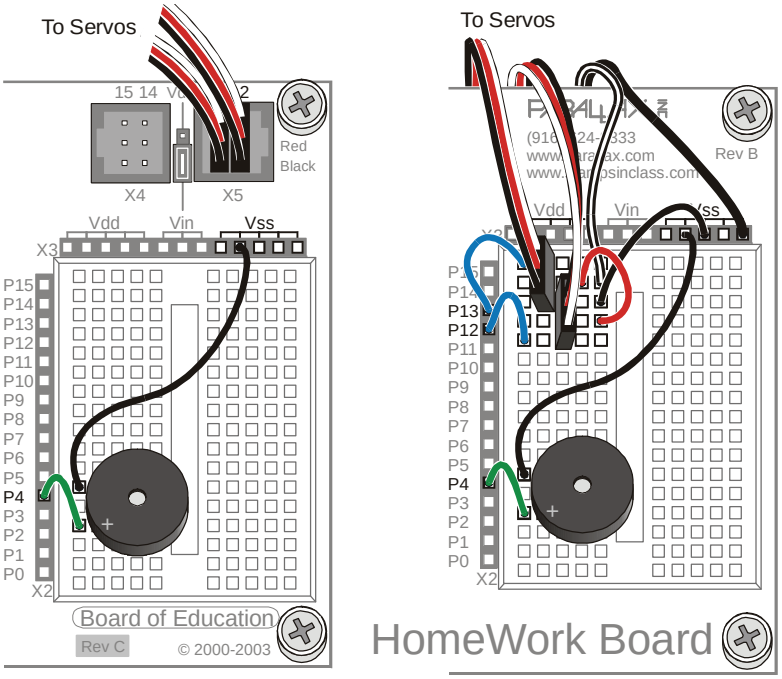


Figure 3-18
Wiring Diagrams for the Program Start/Reset Indicator Circuit

Board of Education (left) and HomeWork Board (right)

Programming the Start/Reset Indicator

The next example program tests the piezospeaker. It uses the **FREQOUT** command to send precisely timed high/low signals to a speaker. Here is the **FREQOUT** command's syntax:

FREQOUT *Pin*, *Duration*, *Freq1* {,*Freq2*}

Here's an example of a **FREQOUT** command that's used in the next example program.

```
FREQOUT 4, 2000, 3000
```

The ***Pin*** argument is 4, meaning that the high/low signals will be sent to I/O pin P4. The ***Duration*** argument, which is how long the high/low signals will last, is 2000, which is 2000 ms or 2 seconds. The ***Freq1*** argument is the frequency of the high/low signals. In this example, the high/low signals will make a 3000 hertz, or 3 kHz, tone.

Frequency can be measured in hertz (Hz). The hertz is a frequency measurement of how many times per second something happens. One hertz is simply one time-per-second, and it's abbreviated 1 Hz. One kilohertz is one-thousand-times-per-second, and it's abbreviated 1 kHz.

FREQOUT digitally synthesizes tones. The **FREQOUT** command applies high/low pulses of varying durations that make a piezospeaker's vibration more closely resemble natural vibrations of music strings.

Example Program: StartResetIndicator.bs2

This example program makes a beep at the beginning of the program, then it goes on to run a program that sends **DEBUG** messages every half second. These messages will continue indefinitely because they are nested between **DO** and **LOOP**. If the power to the BASIC Stamp is interrupted while it is in the middle of its **DO…LOOP**, the program will start at the beginning again. When it starts over, it will beep again. You can simulate a brownout condition by either pressing and releasing the Reset button on your board or disconnecting and reconnecting your board's battery supply.

Learn how to create sound effects and music with your BASIC Stamp! Download *What's a Microcontroller?* from www.parallax.com/go/WAM, and try the example circuit and programs in Chapter 8.

To learn even more about the signals **FREQOUT** generates, download *Understanding Signals with the PropScope* from www.parallax.com/go/PropScope, and read Chapter 7.

- ✓ Reconnect power to your board.
- ✓ Enter, save, and run StartResetIndicator.bs2.
- ✓ Verify that the piezospeaker made a clearly audible tone for two seconds before the "Waiting for reset…" messages started to display in the Debug Terminal.
- ✓ If you did not hear a tone, check your wiring and code for errors. Repeat until you get an audible tone from your speaker.
- ✓ If you did hear an audible tone, try simulating the brownout condition by pressing and releasing the Reset button on your board. Verify that the piezospeaker makes a clearly audible tone after each reset.
- ✓ Also try disconnecting and reconnecting your battery supply, and verify that this results in the reset warning tone as well.

```
' Robotics with the Boe-Bot - StartResetIndicator.bs2
' Test the piezospeaker circuit.
' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.
```

```
DEBUG CLS, "Beep!!!"                          ' Display while speaker beeps.
FREQOUT 4, 2000, 3000                         ' Signal program start/reset.

DO                                            ' DO...LOOP
  DEBUG CR, "Waiting for reset…"              ' Display message
  PAUSE 500                                   ' every 0.5 seconds
LOOP                                          ' until hardware reset.
```


How StartResetIndicator.bs2 Works

StartResetIndicator.bs2 starts by displaying the message "Beep!!!" Then, immediately after printing the message, the **FREQOUT** command plays a 3 kHz tone on the piezoelectric speaker for 2 s. Because the instructions are executed so rapidly by the BASIC Stamp, 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 program enters a **DO…LOOP**, displaying the same "Waiting for reset…" message over and over again. Each time the reset button on the Board of Education is pressed or the power is disconnected and reconnected, the program starts over again, with the "Beep!!!" message and the 3 kHz tone.

Your Turn – Adding StartResetIndicator.bs2 to a Different Program

The **FREQOUT** command in the battery indicator program will be used at the beginning of every example program from here onward. You could consider it part of the "initialization routine" or "boot routine" for every Boe-Bot program.

An initialization routine is comprised of all the commands necessary to get a device or program up and running. It often includes setting certain variable values, beeping noises, and for more complex devices, self testing and calibration.

- ✓ Open HelloOnceEverySecond.bs2.
- ✓ Copy the **FREQOUT** command from the StartResetIndicator.bs2 program into HelloOnceEverySecond.bs2 program, above the **DO…LOOP** section.
- ✓ Run the modified program and verify that it responds with a warning tone every time the BASIC Stamp is reset (either by pressing and releasing the Reset button on the board or disconnecting and reconnecting the battery supply).

ACTIVITY #4: TESTING SPEED CONTROL WITH THE DEBUG TERMINAL

In this activity, you will graph servo speed vs. pulse width. One thing that can make this process go much more quickly is the Debug Terminal's Transmit windowpane, which is shown in Figure 3-19. You can use the Transmit windowpane to send the BASIC Stamp messages. By sending messages that tell the BASIC Stamp what pulse width to deliver to the servo, you can test the servo speed at various pulse widths.

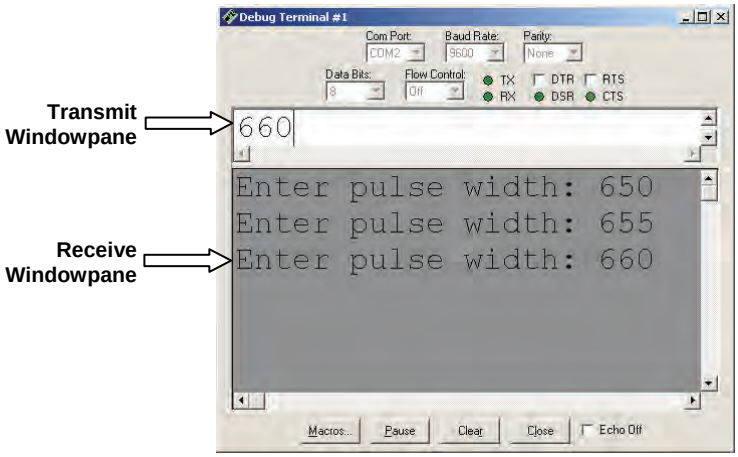


Figure 3-19
Debug Terminal's Transmit and Receive Windowpanes

Pulse width is a common way to describe how long a pulse lasts. The reason it is called pulse "width" is because the amount of time a pulse lasts is related to how wide it is on a timing diagram. Pulses which last longer are wider on timing diagrams and pulses which last for short periods of time are narrow.

Using the DEBUGIN Command

By now, you are probably familiar with the **`DEBUG`** command and how it can be used to send messages from the BASIC Stamp to the Debug Terminal. The place the messages are viewed is called the Receive windowpane because it's the place where messages received from the BASIC Stamp are displayed. The Debug Terminal also has a Transmit windowpane, which allows you to send information to your BASIC Stamp while a program is running. You can use the **`DEBUGIN`** command to make the BASIC Stamp receive what you type into the Transmit windowpane and store it in one or more variables.

The **DEBUGIN** command places the value you type in the Transmit windowpane into a variable. In the next example program, a word variable named **pulseWidth** will be used to store the values the **DEBUGIN** command receives.

```
pulseWidth    VAR    Word
```

Now, the **DEBUGIN** command can be used to capture a decimal value that you enter into the Debug Terminal's Transmit windowpane and store it in **pulseWidth**:

```
DEBUGIN DEC pulseWidth
```

You can then program the BASIC Stamp to use this value. Here it is used in the **PULSOUT** command's ***Duration*** argument:

```
PULSOUT 12, pulseWidth
```

Example Program: TestServoSpeed.bs2

This program allows you to set the **PULSOUT** command's ***Duration*** argument by entering it into the Debug Terminal's Transmit windowpane.

- ✓ Continue this activity with the Boe-Bot sitting on its nose so that the wheels do not touch the ground.
- ✓ Enter, save, and run TestServoSpeed.bs2.
- ✓ Point at the Debug Terminal's Transmit windowpane with your mouse, and click it to activate the cursor in that window for typing.
- ✓ Type 650 and then press the Enter key.
- ✓ Verify that the servo turns full speed clockwise for six seconds.

When the servo is done turning, you will be prompted to enter another value.

- ✓ Type 850 and then press the Enter key.
- ✓ Verify that the servo turns full speed counterclockwise.

Try measuring the wheel's rotational speed in RPM (revolutions per minute) for a range of pulse widths between 650 and 850. Here's how:

- ✓ Place a mark on the wheel so that you can see how far it turns in 6 seconds.

- ✓ Use the Debug Terminal to test how far the wheel turns for each of these pulse widths: 650, 660, 670, 680, 690, 700, 700, 710, 720, 730, 740, 750, 760, 770, 780, 790, 800, 810, 820, 830, 840, 850
- ✓ For each pulse width, multiply the number of turns by 10 to get the RPM. For example, if the wheel makes 3.65 full turns, it was rotating at 36.5 RPM.
- ✓ Explain in your own words how you can use pulse width to control Continuous Rotation servo speed.

```
' Robotics with the Boe-Bot - TestServoSpeed.bs2
' Enter pulse width, then count revolutions of the wheel.
' The wheel will run for 6 seconds
' Multiply by 10 to get revolutions per minute (RPM).

'{$STAMP BS2}
'{$PBASIC 2.5}

counter        VAR     Word
pulseWidth     VAR     Word
pulseWidthComp VAR     Word

FREQOUT 4, 2000, 3000                        ' Signal program start/reset.

DO

  DEBUG "Enter pulse width: "

  DEBUGIN DEC pulseWidth

  pulseWidthComp = 1500 - pulseWidth

  FOR counter = 1 TO 244
    PULSOUT 12, pulseWidth
    PULSOUT 13, pulseWidthComp
    PAUSE 20
  NEXT

LOOP
```

How TestServoSpeed.bs2 Works

Three variables are declared, **`counter`** for the **`FOR…NEXT`** loop, **`pulseWidth`** for the **`DEBUGIN`** and **`PULSOUT`** commands, and **`pulseWidthComp`** which stores a value that is used in a second **`PULSOUT`** command.

```
counter        VAR     Word
pulseWidth     VAR     Word
pulseWidthComp VAR     Word
```

The **FREQOUT** command is used to signal that the program has started.

```
FREQOUT 4,2000,3000
```

The remainder of the program is nested within a **DO…LOOP**, so it will execute over and over again. The Debug Terminal's operator (that's you) is asked to enter a pulse width. The **DEBUGIN** command stores this value in the **pulseWidth** variable.

```
DEBUG "Enter pulse width: "

DEBUGIN DEC pulseWidth
```

To make the measurement more accurate, two **PULSOUT** commands have to be sent. By making one **PULSOUT** command the same amount below 750 as the other is above 750, the sum of the two **PULSOUT** ***Duration*** arguments is always 1500. That ensures that the two **PULSOUT** commands combined take the same amount of time. The result is that no matter the ***Duration*** of your **PULSOUT** command, the **FOR…NEXT** loop will still take the same amount of time to execute. This will make the RPM measurements you will take in the Your Turn section more accurate.

This next command takes the pulse width you entered, and calculates a pulse width that will make 1500 when the two are added together. If you enter a pulse width of 650, **pulseWidthComp** will be 850. If you enter a pulse width of 850, **pulseWidthComp** will be 650. If you enter a pulse width of 700, **pulseWidthComp** will be 800. Try a few other examples. They will all add up to 1500.

```
pulseWidthComp = 1500 - pulseWidth
```

A **FOR…NEXT** loop that runs for 6 seconds sends pulses to the right (P12) servo. The **pulseWidthComp** value is sent to the left (P13) servo, making it turn in the opposite direction.

```
FOR counter = 1 TO 244
  PULSOUT 12, pulseWidth
  PULSOUT 13, pulseWidthComp
  PAUSE 20
NEXT
```

Your Turn – Advanced Topic: Graphing Pulse Width vs. Rotational Velocity

Figure 3-20 shows an example of a transfer curve for a continuous rotation servo. The horizontal axis shows the pulse width in ms, and the vertical axis shows the rotational

velocity in RPM. In this graph, clockwise is negative and counterclockwise is positive. This particular servo's transfer curve ranges from about -48 RPM to 48 RPM over the range of test pulse widths that range from 1.3 ms to 1.7 ms.

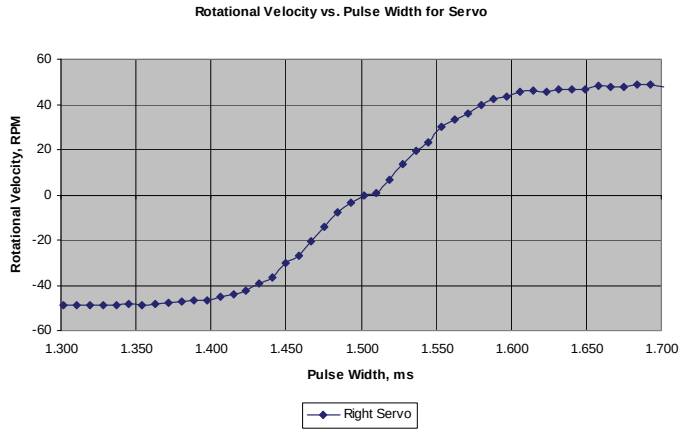


Figure 3-20
Transfer Curve Example for Parallax Continuous Rotation Servo

Remember that the **PULSOUT** command's ***Duration*** argument is in 2 µs units. **PULSOUT 12, 650** sends pulses that last 1.3 ms to P12. **PULSOUT 12, 655** sends pulses of 1.31 ms, **PULSOUT 12, 660** sends pulses of 1.32 ms, and so on.

$$Duration = 650 \times 2\ \mu s = 650 \times 0.000002\ s = 0.0013\ s = 1.3\ ms$$

$$Duration = 655 \times 2\ \mu s = 655 \times 0.000002\ s = 0.00131\ s = 1.31\ ms$$

$$Duration = 660 \times 2\ \mu s = 660 \times 0.000002\ s = 0.00132\ s = 1.32\ ms$$

You can use Table 3-1 to record the data for your own transfer curve. Keep in mind that the example program is controlling the right wheel with the values you enter. The left wheel turns in the opposite direction.

- ✓ Mark your right wheel so that you have a reference point to count the revolutions.
- ✓ Run TestServoSpeed.bs2.

Table 3-1: Pulse Width and RPM for Parallax Continuous Rotation Servo							
Pulse Width (ms)	**Rotational Velocity (RPM)**	**Pulse Width (ms)**	**Rotational Velocity (RPM)**	**Pulse Width (ms)**	**Rotational Velocity (RPM)**	**Pulse Width (ms)**	**Rotational Velocity (RPM)**
1.300		1.400		1.500		1.600	
1.310		1.410		1.510		1.610	
1.320		1.420		1.520		1.620	
1.330		1.430		1.530		1.630	
1.340		1.440		1.540		1.640	
1.350		1.450		1.550		1.650	
1.360		1.460		1.560		1.660	
1.370		1.470		1.570		1.670	
1.380		1.480		1.580		1.680	
1.390		1.490		1.590		1.690	
						1.700	

- ✓ Click the Debug Terminal's Transmit windowpane.
- ✓ Enter the value 650.
- ✓ Count how many turns the wheel made.

Since the servo turns for 6 seconds, you can multiply this value by 10 to get revolutions per minute (RPM).

- ✓ Multiply this value by 10 and enter the result next to the 1.3 ms table entry.
- ✓ Enter the value 655, and count how many turns the wheel made.
- ✓ Multiply this value by 10 and enter the result next to the 1.31 ms table entry.
- ✓ Keep increasing your durations by 5 (0.01 ms) until you are up to 850 (1.7 ms).
- ✓ Use a spreadsheet, calculator, or graph paper to graph the data.
- ✓ Repeat this process for your other servo.

To repeat these measurements for the left wheel, modify the **`PULSOUT`** commands so that pulses with a ***Duration*** of **`pulseWidth`** are sent to P13 and pulses with a ***Duration*** of **`pulseWidthComp`** are sent to P12.

SUMMARY

This chapter covered Boe-Bot assembly and testing. This involved mechanical assembly, such as connecting the various moving parts to the Boe-Bot chassis. It also involved circuit assembly, connecting the servos and piezospeaker. The testing involved retesting the servos after they were disconnected to build the Boe-Bot.

The concept of brownout was introduced along with what this condition does to a program running on the BASIC Stamp. Brownout causes the BASIC Stamp to shut down, and then start running the program from the beginning. A piezospeaker was added to signal the start of a program. If the piezospeaker sounds in the middle of a running program when it's not supposed to, this can indicate a brownout condition. Brownout conditions can in turn indicate low batteries. To make the piezospeaker play a tone to indicate a reset, the **`FREQOUT`** command was introduced. This command is part of an initialization routine that will be used at the beginning of all Boe-Bot programs.

Until this chapter, the Debug Terminal has been used to display messages sent to the computer by the BASIC Stamp. These messages were displayed in the Receive windowpane. The Debug Terminal also has a Transmit windowpane that you can use to send values to the BASIC Stamp. The BASIC Stamp can capture these values by executing the **`DEBUGIN`** command, which receives a value sent by the Debug Terminal's Transmit windowpane and stores it in a variable. The value can then be used by the PBASIC program. This technique was used to set the pulse widths to control and test servo speed and direction. It was also used as a data collection aid for plotting the transfer curve of a Parallax Continuous Rotation servo.

Questions

1. What are some of the symptoms of brownout on the Boe-Bot?
2. How can a piezospeaker be used to detect brownout?
3. What is a reset?
4. What is an initialization routine?
5. What are three (or more) possible mistakes that can occur when disconnecting and reconnecting the servos?
6. What command do you have to change in RightServoTest.bs2 to test the left wheel instead of the right wheel?

Exercises

1. Write a **`FREQOUT`** command that makes a tone that sounds different from the reset detect tone to signify the end of a program.
2. Write a **`FREQOUT`** command that makes a tone (different from beginning or ending tones) that signifies an intermediate step in a program has been completed. Try a value with a 100 ms duration at a 4 kHz frequency.

Projects

1. Modify RightServoTest.bs2 so that it makes a tone signifying the test is complete.
2. Modify TestServoSpeed.bs2 so that you can use **`DEBUGIN`** to enter the pulse width for the left and the right servo as well as the number of pulses to deliver in the **`FOR…NEXT`** loop. Use this program to control your Boe-Bot's motion via the Debug Terminal's Transmit windowpane.

Solutions

Q1. Symptoms include erratic behavior such as going in unexpected directions or doing a confused dance.

Q2. A **FREQOUT** command at the beginning of all Boe-Bot programs causes the piezospeaker to play a tone. This tone will therefore occur every time an accidental reset happens due to brownout conditions.

Q3. A reset is when the power is interrupted and the BASIC Stamp program starts running again from the beginning of the program.

Q4. An initialization routine consists of the lines of code that are used at the beginning of the program. These lines of code run each time the program starts from the beginning.

Q5. 1) The servo lines P12 and P13 are swapped. 2) One or both servos is plugged in backwards, so that the white-red-black color coding is incorrect. 3) The power switch is not on position-2. 4) The 9V or AA batteries are not installed. 5) The servo centering potentiometer is out of adjustment.

Q6. The **PULSOUT** commands must be changed to read **PULSOUT 13** instead of **PULSOUT 12**.

E1. The key is to modify the **FREQOUT** command used for the StartResetIndicator.bs2 program, that is, **FREQOUT, 4, 2000, 3000**. For example: **FREQOUT, 4, 500, 3500** would work.

E2. **FREQOUT 4, 100, 4000**.

P1. The key to solving this program is to add the line from Exercise 1 above the **END** command in the RightServoTest.bs2 program.

```
' Robotics with the Boe-Bot - Ch03Prj01_TestCompleteTone.bs2
' Right servo turns clockwise three seconds, stops 1 second, then
' counterclockwise three seconds.   A tone signifies that the
' test is complete.

' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Program Running!"

counter        VAR     Word

FREQOUT 4, 2000, 3000              ' Signal start of program.

FOR counter = 1 TO 122             ' Clockwise just under 3 seconds.
  PULSOUT 12, 650
  PAUSE 20
NEXT

FOR counter = 1 TO 40              ' Stop one second.
```

```
  PULSOUT 12, 750
  PAUSE 20
NEXT

FOR counter = 1 TO 122             ' Counterclockwise three seconds.
  PULSOUT 12, 850
  PAUSE 20
NEXT

FREQOUT 4, 500, 3500               ' Signal end of program

END
```

P2. To solve this problem, TestServoSpeed.bs2 must be expanded to receive three pieces of data: left servo pulsewidth, right servo pulsewidth, and number of pulses. Then, a **FOR…NEXT** loop with two servo **PULSOUT** commands must be added to actually move the servo motors. Furthermore, all variables must be declared in the beginning of the program. An example solution is shown below. Note: This project is best tested with the Boe-Bot's wheels propped up.

```
' Robotics with the Boe-Bot - Ch03Prj02_DebuginMotion.bs2
' Enter servo pulsewidth & duration for both wheels via Debug Terminal.

'{$STAMP BS2}
'{$PBASIC 2.5}

ltPulseWidth   VAR     Word            ' Left servo pulse width
rtPulseWidth   VAR     Word            ' Right servo pulse width
pulseCount     VAR     Byte            ' Number of pulses to servo
counter        VAR     Word            ' Loop counter

DO
  DEBUG "Enter left servo pulse width: "  ' Enter values in Debug
  DEBUGIN DEC ltPulseWidth                ' Terminal

  DEBUG "Enter right servo pulse width: "
  DEBUGIN DEC rtPulseWidth

  DEBUG "Enter number of pulses:        "
  DEBUGIN DEC pulseCount

  FOR counter = 1 TO pulseCount        ' Send specific number of pulses
    PULSOUT 13, ltPulseWidth           ' Left servo motion
    PULSOUT 12, rtPulseWidth           ' Right servo motion
    PAUSE 20
  NEXT

LOOP
```

Chapter 4: Boe-Bot Navigation

The Boe-Bot can be programmed to perform a variety of maneuvers. The maneuvers and programming techniques introduced in this chapter will be reused in later chapters. The only difference is that in this chapter, the Boe-Bot will blindly perform the maneuvers. In later chapters, the Boe-Bot will perform similar maneuvers in response to conditions it detects with its sensors.

This chapter also introduces ways to tune and calibrate the Boe-Bot's navigation. Included are techniques to straighten a Boe-Bot's forward drive, more precise turns, and calculating distances.

Activity	Summary
1	Program the Boe-Bot to perform the basic maneuvers: forward, backward, rotate left, rotate right, and pivoting turns.
2	Tune the maneuvers from Activity #1 so that they are more precise.
3	Use math to calculate the number of pulses to deliver to make the Boe-Bot travel a predetermined distance.
4	Instead of programming the Boe-Bot to make abrupt starts and stops, write programs that make the Boe-Bot gradually accelerate into and decelerate out of maneuvers.
5	Write subroutines to perform the basic maneuvers so that each subroutine can be used over and over again in a program.
6	Record complex maneuvers in the BASIC Stamp module's unused program memory and write programs that play back these maneuvers.

ACTIVITY #1: BASIC BOE-BOT MANEUVERS

Figure 4-1 shows your Boe-Bot's front, back, left, and right. When the Boe-Bot goes forward, in the picture, it would have to roll to the right edge of the page. Backward would be toward the left edge of the page. A left turn would be make the Boe-Bot ready to drive off the top of the page, and a right turn would have it facing the bottom of the page.

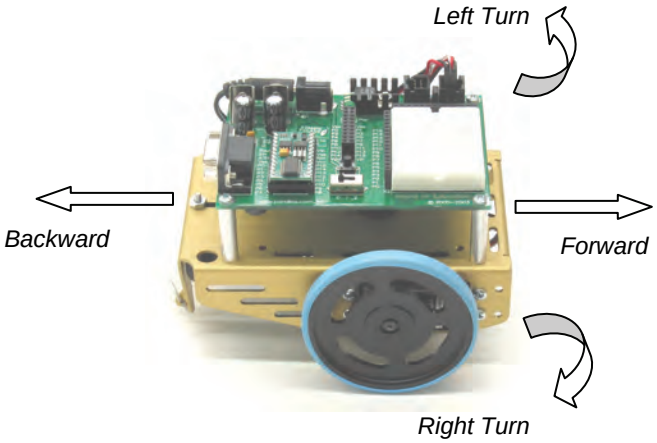


Figure 4-1
Your Boe-Bot and Driving Directions

Moving Forward

Here's a funny thing: to make the Boe-Bot go forward, the Boe-Bot's left wheel has to turn counterclockwise, but its right wheel has to turn clockwise. If you haven't already grasped this, take a look at Figure 4-2 and see if you can convince yourself that it's true. Viewed from the left, the wheel has to turn counterclockwise for the Boe-Bot to move forward. Viewed from the right, the other wheel has to turn clockwise for the Boe-Bot to move forward.

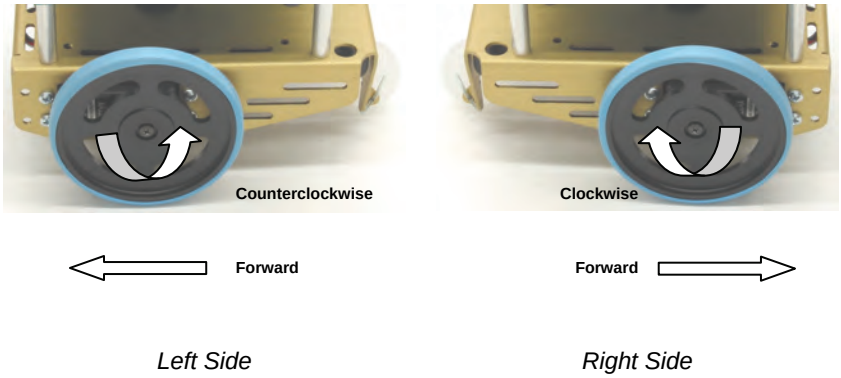


Figure 4-2
Wheel Rotation for Forward Motion

Remember from Chapter 2 that the **PULSOUT** command's ***Duration*** argument controls the speed and direction the servo turns. The ***StartValue*** and ***EndValue*** arguments of a **FOR…NEXT** loop control the number of pulses that are delivered. Since each pulse takes

the same amount of time, the ***EndValue*** argument also controls the time the servo runs. Here's an example program that will make the Boe-Bot roll forward for about three seconds.

Example Program: BoeBotForwardThreeSeconds.bs2

- ✓ Make sure power is connected to the BASIC Stamp and servos.
- ✓ Enter, save, and run BoeBotForwardThreeSeconds.bs2.


```
' Robotics with the Boe-Bot - BoeBotForwardThreeSeconds.bs2
' Make the Boe-Bot roll forward for three seconds.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

counter        VAR     Word

FREQOUT 4, 2000, 3000                        ' Signal program start/reset.

FOR counter = 1 TO 122                       ' Run servos for 3 seconds.

  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20

NEXT

END
```

How BoeBotForwardThreeSeconds.bs2 Works

From chapter 2, you already have lots of experience with the elements of this program: a variable declaration, a **FOR…NEXT** loop, **PULSOUT** commands with ***Pin*** and ***Duration*** arguments, and **PAUSE** commands. Here's a review of what each does and how it relates to the servos' motions.

First a variable is declared that will be used in the **FOR...NEXT** loop.

```
counter VAR Word
```

You should recognize this next command; it generates a tone to signal the start of the program. It will be used in all programs that run the servos.

```
FREQOUT 4, 2000, 3000            ' Signal program start/reset.
```

This **FOR…NEXT** loop sends 122 sets of pulses to the servos, one each to P13 and P12, pausing for 20 ms after each set and then returning to the top of the loop.

```
FOR counter = 1 TO 122
  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20
NEXT
```

PULSOUT 13, 850 causes the left servo to rotate counterclockwise while **PULSOUT 12, 650** causes the right servo to rotate clockwise. Therefore, both wheels will be turning toward the front end of the Boe-Bot, causing it to drive forward. It takes about 3 seconds for the **FOR…NEXT** loop to execute 122 times, so the Boe-Bot drives forward for about 3 seconds.

Your Turn – Adjusting Distance and Speed

- ✓ By changing the **FOR…NEXT** loop's ***EndValue*** argument from 122 to 61, you can make the Boe-Bot move forward for half the time. This in turn will make the Boe-Bot move forward half the distance.
- ✓ Save BoeBotForwardThreeSeconds.bs2 under a new name.
- ✓ Change the **FOR...NEXT** loop's ***EndValue*** from 122 to 61.
- ✓ Run the program and verify that it ran at half the time and covered half the distance.
- ✓ Try these steps over again, but this time, change the **FOR…NEXT** loop's ***EndValue*** to 244.

The **PULSOUT** ***Duration*** arguments of 650 and 850 caused the servos to rotate near their maximum speed. By bringing each of the **PULSOUT** ***Duration*** arguments closer to the stay-still value of 750, you can slow down your Boe-Bot.

- ✓ Modify your program with these **PULSOUT** commands:

```
PULSOUT 13, 780
PULSOUT 12, 720
```

- ✓ Run the program, and verify that your Boe-Bot moves slower.

Moving Backward, Rotating, and Pivoting

All it takes to get other motions out of your Boe-Bot are different combinations of the **PULSOUT** ***Duration*** arguments. For example, these two **PULSOUT** commands can be used to make your Boe-Bot go backwards:

```
PULSOUT 13, 650
PULSOUT 12, 850
```

These two commands will make your Boe-Bot rotate in a left turn (counterclockwise as you are looking at it from above):

```
PULSOUT 13, 650
PULSOUT 12, 650
```

These two commands will make your Boe-Bot rotate in a right turn (clockwise as you are looking at it from above):

```
PULSOUT 13, 850
PULSOUT 12, 850
```

You can combine all these commands into a single program that makes the Boe-Bot move forward, turn left, turn right, then move backward.

Example Program: ForwardLeftRightBackward.bs2

- ✓ Enter, save, and run ForwardLeftRightBackward.bs2.

TIP – To enter this program quickly, use the BASIC Stamp Editor's Edit menu tools (Copy and Paste) to make four copies of a **FOR…NEXT** loop. Then, adjust only the **PULSOUT** ***Duration*** values and **FOR…NEXT** loop ***EndValues***.

```
' Robotics with the Boe-Bot - ForwardLeftRightBackward.bs2
' Move forward, left, right, then backward for testing and tuning.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

counter        VAR     Word

FREQOUT 4, 2000, 3000                        ' Signal program start/reset.
```

```
FOR counter = 1 TO 64                         ' Forward

  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20

NEXT

PAUSE 200

FOR counter = 1 TO 24                         ' Rotate left - about 1/4 turn

  PULSOUT 13, 650
  PULSOUT 12, 650
  PAUSE 20

NEXT

PAUSE 200

FOR counter = 1 TO 24                         ' Rotate right - about 1/4 turn

  PULSOUT 13, 850
  PULSOUT 12, 850
  PAUSE 20

NEXT

PAUSE 200

FOR counter = 1 TO 64                         ' Backward

  PULSOUT 13, 650
  PULSOUT 12, 850
  PAUSE 20

NEXT

END
```

Your Turn – Pivoting

You can make the Boe-Bot turn by pivoting around one wheel. The trick is to keep one wheel still while the other rotates. For example, if you keep the left wheel still and make the right wheel turn clockwise (forward), the Boe-Bot will pivot to the left.

```
PULSOUT 13, 750
PULSOUT 12, 650
```

If you want to pivot forward and to the right, simply stop the right wheel, and make the left wheel turn counterclockwise (forward).

```
PULSOUT 13, 850
PULSOUT 12, 750
```

These are the **PULSOUT** commands for pivoting backwards and to the right.


```
PULSOUT 13, 650
PULSOUT 12, 750
```

Finally, these are the **PULSOUT** commands for pivoting backwards and to the left.

```
PULSOUT 13, 750
PULSOUT 12, 850
```

- ✓ Save ForwardLeftRightBackward.bs2 as PivotTests.bs2.
- ✓ Substitute the **PULSOUT** commands just discussed in place of the forward, left, right, and backward routines.
- ✓ Adjust the run time of each maneuver by changing each **FOR…NEXT** loop's ***EndValue*** to 30.
- ✓ Be sure to change the comment next to each **FOR…NEXT** loop to reflect each new pivot action.
- ✓ Run the modified program and verify that the different pivot actions work.

ACTIVITY #2: TUNING THE BASIC MANEUVERS

Imagine writing a program that instructs the Boe-Bot to travel full-speed forward for fifteen seconds. What if the Boe-Bot 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-Bot back apart and re-adjust the servos with a screwdriver to fix this. You can simply adjust the program slightly to get both Boe-Bot wheels traveling the same speed. While the screwdriver approach would be called a "hardware adjustment," the programming approach is called a "software adjustment."

Straightening the Boe-Bot's Path

The first step is to examine your Boe-Bot's travel for long enough to find out if it's curving either to the left or to the right when it's supposed to be going straight ahead. Ten seconds of forward travel should be enough. This can be accomplished with a simple modification to BoeBotForwardThreeSeconds.bs2 from the previous activity.

Example Program: BoeBotForwardTenSeconds.bs2

- ✓ Open BoeBotForwardThreeSeconds.bs2.
- ✓ Rename and save it as BoeBotForwardTenSeconds.bs2.
- ✓ Change the **`EndValue`** of the **`FOR counter`** from 122 to 407, so it reads like this:

```
' Robotics with the Boe-Bot - BoeBotForwardTenSeconds.bs2
' Make the Boe-Bot roll forward for ten seconds.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

counter        VAR     Word

FREQOUT 4, 2000, 3000                         ' Signal program start/reset.

FOR counter = 1 TO 407                        ' Number of pulses – run time.

  PULSOUT 13, 850                             ' Left servo full speed ccw.
  PULSOUT 12, 650                             ' Right servo full speed cw.
  PAUSE 20

NEXT

END
```

- ✓ Run the program, and watch closely to see if your Boe-Bot veers to the right or left as it travels forwards for ten seconds.

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

If your Boe-Bot goes perfectly straight, try this example anyway. If you follow the instructions, it should adjust your Boe-Bot so that it curves slightly to the right.

Let's say that the Boe-Bot turns slightly to the left. There are two ways to think about this problem: either the left wheel is turning too slowly, or the right wheel is turning too quickly. Since the Boe-Bot is already at full speed, speeding up the left wheel isn't going to be practical, but slowing down the right wheel should help remedy the situation.

Remember that servo speed is determined by the **`PULSOUT`** command's ***Duration*** argument. The closer the ***Duration*** is to 750, the slower the servo turns. This means you should change the 650 in the command **`PULSOUT 12,650`** to something a little closer

to 750. If the Boe-Bot is only just a little off course, maybe **PULSOUT 12,663** will do the trick. If the servos are severely mismatched, maybe it needs to be **PULSOUT 12,690**.

It will probably take several tries to get the right value. Let's say that your first guess is that **PULSOUT 12,663** will do the trick, but it turns out not to be enough because the Boe-Bot is still turning slightly to the left. So try **PULSOUT 12,670**. Maybe that overcorrects, and it turns out that **PULSOUT 12,665** gets it exactly right. This is called an iterative process, meaning a process that takes repeated tries and refinements to get to the right value.

If your Boe-Bot curved to the right instead of the left, it means you need to slow down the left wheel by reducing the ***Duration*** of 850 in the **PULSOUT 13,850** command. Again, the closer this value gets to 750, the slower the servo will turn.

- ✓ Modify BoeBotForwardTenSeconds.bs2 so that it makes your Boe-Bot go straight forward.
- ✓ Use masking tape or a sticker to label each servo with the best **PULSOUT** values.
- ✓ If your Boe-Bot already travels straight forward, try the modifications just discussed to see the effect. It should cause the Boe-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-Bot to roll backward.

- ✓ Modify BoeBotForwardTenSeconds.bs2 so that it makes the Boe-Bot roll backward for ten seconds.
- ✓ Repeat the test for straight line.
- ✓ Repeat the steps for correcting the **PULSOUT** command's ***Duration*** argument to straighten the Boe-Bot's backward travel.

Tuning the Turns

Software adjustments can also be made to get the Boe-Bot to turn to a desired angle, such as 90°. The amount of time the Boe-Bot spends rotating in place determines how far it turns. Because the **FOR…NEXT** loop controls run time, you can adjust the **FOR…NEXT** loop's ***EndValue*** argument to get very close to the turning angle you want.

Here's the left turn routine from ForwardLeftRightBackward.bs2:

```
FOR counter = 1 TO 24         ' Rotate left - about 1/4 turn

  PULSOUT 13, 650
  PULSOUT 12, 650
  PAUSE 20

NEXT
```

Let's say that the Boe-Bot turns just a bit more than 90° (1/4 of a full circle). Try **`FOR counter = 1 TO 23`**, or maybe even **`FOR counter = 1 TO 22`**. If it doesn't turn far enough, increase the run time of the rotation by increasing the **`FOR…NEXT`** loop's ***EndValue*** argument to whatever value it takes to complete the quarter turn.

If you find yourself with one value slightly overshooting 90° and the other slightly undershooting, try choosing the value that makes it turn a little too far, then slow down the servos slightly. In the case of the rotate left, both **`PULSOUT`** ***Duration*** arguments should be changed from 650 to something a little closer to 750. As with the straight line exercise, this will also be an iterative process.

Your Turn – 90° Turns

- ✓ Modify ForwardLeftRightBackward.bs2 so that it makes precise 90° turns.
- ✓ Update ForwardLeftRightBackward.bs2 with the **`PULSOUT`** values that you determined for straight forward and backward travel.
- ✓ Update the label on each servo with a notation about the appropriate ***EndValue*** for a 90° turn.

Carpeting can cause navigation errors. If you are running your Boe-Bot on carpeting, don't expect perfect results! A carpet is a bit like a golf green—the way the carpet pile is inclined can affect the way your Boe-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 lends itself to 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 speedometer in your car. 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:

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

Example – Time for English Distance

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.

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

Example – Time for Metric Distance

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.

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

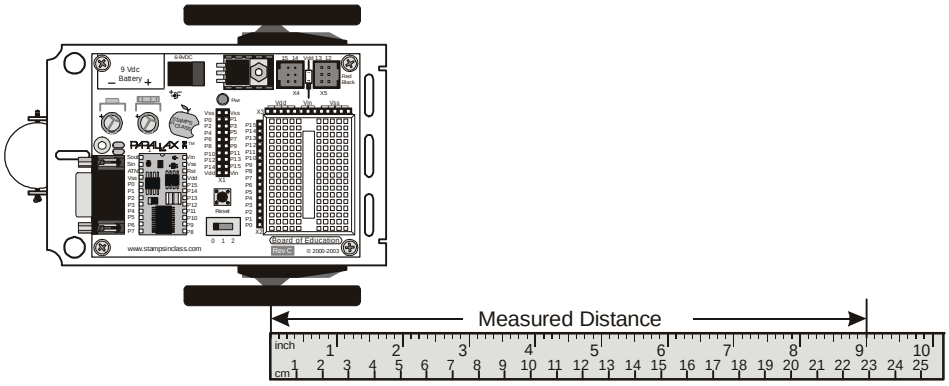
You can do the same exercise with the Boe-Bot, except you have control over how far away the destination is. Here's the equation you will use:

$$servo\ run\ time = \frac{Boe\text{-}Bot\ distance}{Boe\text{-}Bot\ speed}$$

You will have to test the Boe-Bot speed. The easiest way to do this is to set the Boe-Bot next to a ruler and make it travel forward for one second. By measuring how far your Boe-Bot traveled, you will know your Boe-Bot's speed. If your ruler has inches, your answer will be in inches per second (in/s), if it has centimeters your answer will be in centimeters per second (cm/s).

- ✓ Enter, save, and run ForwardOneSecond.bs2.
- ✓ Place your Boe-Bot next to a ruler as shown in Figure 4-3.
- ✓ Make sure to line up the point where the wheel touches the ground with the 0 in/cm mark on the ruler.

Figure 4-3: Measuring Boe-Bot Distance



- ✓ Press the Reset button on your board to re-run the program.
- ✓ Measure how far your Boe-Bot traveled by recording the measurement where the wheel is now touching the ground here:_____ in / cm.

```
' Example Program: ForwardOneSecond.bs2
' Robotics with the Boe-Bot - ForwardOneSecond.bs2
' Make the Boe-Bot roll forward for one second.
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"
counter        VAR     Word

FREQOUT 4, 2000, 3000                          ' Signal program start/reset.

FOR counter = 1 TO 41

  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20
NEXT

END
```

You can also think about the distance you just recorded as your Boe-Bot's speed, in units per second. Let's say that your Boe-Bot traveled 9 in (23 cm). Since it took one second for your Boe-Bot to travel that far, it means your Boe-Bot travels at around 9 in/s (23 cm/s). Now, you can figure out how many seconds your Boe-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-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.

Example – Time for 20 Inches

At 9 in/s, your Boe-Bot has to travel for 2.22 s to travel 20 in.

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

Example – Time for 51 Centimeters

At 23 cm/s, your Boe-Bot has to travel for 2.22 s to travel 51 cm.

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

In Chapter 2, Activity #6, we learned that it takes 24.6 ms (0.024 s) each time the two servo **PULSOUT** and one **PAUSE** commands are executed in a **FOR…NEXT** loop. The reciprocal of this value is the number of pulses per second that the loop transmits to each servo. A reciprocal is when you swap a fraction's numerator and denominator. Another way to take a reciprocal is to divide a number or fraction into the number one. In other words, 1 ÷ 0.024 s/pulse = 40.65 pulses/s.

Since you know the amount of time you want your Boe-Bot to move forward (2.22 s) and the number of pulses the BASIC Stamp sends to the servos each second (40.65 pulses/s), you can use these values to calculate how many pulses to send to the servos. This is the number you will have to use for your **FOR…NEXT** loop's ***EndValue*** argument.

$$\begin{aligned} pulses &= 2.22 s \times \frac{40.65\ pulses}{s} \\ &= 90.24...\ pulses \\ &\approx 90\ pulses \end{aligned}$$

The calculations in this example took two steps. First, figure out how long the servos have to run to make the Boe-Bot travel a certain distance, then figure out how many pulses it takes to make the servos run for that long. Since you know you have to multiply by 40.65 to get from run time to pulses, you can reduce this to one step.

$$pulses = \frac{Boe-Bot\ distance}{Boe-Bot\ speed} \times \frac{40.65\ pulses}{s}$$

Example – Time for 20 Inches

At 9 in/s, your Boe-Bot has to travel for 2.22 s to travel 20 in.

$$\begin{aligned} pulses &= \frac{20\ in}{9\ in/s} \times \frac{40.65\ pulses}{s} \\ &= 20\ in \times \frac{1\ s}{9\ in} \times \frac{40.65\ pulses}{1\ s} \\ &= 20 \div 9 \times 40.65\ pulses \\ &= 90.333...pulses \\ &\approx 90\ pulses \end{aligned}$$

Example – Time for 51 Centimeters

At 23 cm/s, your Boe-Bot has to travel for 2.22 s to travel 51 cm.

$$\begin{aligned} pulses &= \frac{51\ cm}{23\ cm/s} \times \frac{40.65\ pulses}{s} \\ &= 51cm \times \frac{1\ s}{23cm} \times \frac{40.65\ pulses}{1s} \\ &= 51 \div 23 \times 40.65\ pulses \\ &= 90.136...pulses \\ &\approx 90\ pulses \end{aligned}$$

Your Turn – Your Boe-Bot's Distance

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

- ✓ If you have not already done so, use a ruler and the ForwardOneSecond.bs2 program to determine your Boe-Bot's speed in in/s or cm/s.
- ✓ Decide how far you want your Boe-Bot to travel.
- ✓ Use the pulses equation to figure out how many pulses to deliver to the Boe-Bot's servos:

$$pulses = \frac{Boe-Bot\ distance}{Boe-Bot\ speed} \times \frac{40.65\ pulses}{s}$$

- ✓ Modify BoeBotForwardOneSecond.bs2 so that it delivers the number of pulses you determined for your distance.
- ✓ Run the program and test to see how close you got.

This technique has sources of error. The activity you just completed does not take into account the fact that it took a certain number of pulses for the Boe-Bot to get up to full speed. Nor did it take into account any distance the Boe-Bot might coast before it comes to a full stop. The servo speeds will also go slower as the batteries lose their charge.

You can increase the accuracy of your Boe-Bot distances with devices called encoders, which count the holes in the Boe-Bot's wheels as they pass. Encoder kits and other Boe-Bot specific accessories are available from www.parallax.com/go/Boe-Bot.

ACTIVITY #4: MANEUVERS—RAMPING

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-Bot's batteries and your servos.

Programming for Ramping

The key to ramping is to use variables along with constants for the **`PULSOUT`** command's **`Duration`** argument. Figure 4-4 shows a **`FOR…NEXT`** loop that can ramp the Boe-Bot's speed from full stop to full speed ahead. Each time the **`FOR…NEXT`** loop repeats itself, the **`pulseCount`** variable increases by 1. The first time through, **`pulseCount`** is 1, so it's like using the commands **`PULSOUT 13, 751`** and **`PULSOUT 12, 749`**. The second time through the loop, the value of **`pulseCount`** is 2, so it's like using the commands **`PULSOUT 13, 752`** and **`PULSOUT 12, 748`**. As the value of the **`pulseCount`** variable increases, so does the speed of the servos. By the hundredth time through the loop, the **`pulseCount`** variable is 100, so it's like using the commands **`PULSOUT 13, 850`** and **`PULSOUT 12, 650`**, which is full-speed ahead for the Boe-Bot.

```
pulseCount    VAR     Word

FOR pulseCount = 1 TO 100

  PULSOUT 13, 750 + pulseCount
  PULSOUT 12, 750 - pulseCount
  PAUSE 20

NEXT
```

1, 2, 3, …100

Figure 4-4
Ramping Example

Recall from Chapter 2, Activity #5 that **FOR…NEXT** loops can also count downward from a higher number to a lower number. You can use this to ramp the speed back down again by using **FOR pulseCount = 100 TO 1**. Here is an example program that uses **FOR…NEXT** loops to ramp up to full speed, then ramp back down.

Example Program: StartAndStopWithRamping.bs2

- ✓ Enter, save, and run StartAndStopWithRamping.bs2.
- ✓ Verify that the Boe-Bot gradually accelerates to full speed, maintains full speed for a while, and then gradually decelerates to a full stop.

```
' -----[ Title ]-----------------------------------------------------------
' Robotics with the Boe-Bot - StartAndStopWithRamping.bs2
' Ramp up, go forward, ramp down.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

pulseCount    VAR     Word                  ' FOR...NEXT loop counter.

' -----[ Initialization ]---------------------------------------------------

FREQOUT 4, 2000, 3000                       ' Signal program start/reset.

' -----[ Main Routine ]-----------------------------------------------------

' Ramp up forward.

FOR pulseCount = 1 TO 100                   ' Loop ramps up for 100 pulses.
  PULSOUT 13, 750 + pulseCount              ' Pulse = 1.5 ms + pulseCount.
  PULSOUT 12, 750 - pulseCount              ' Pulse = 1.5 ms – pulseCount.
  PAUSE 20                                  ' Pause for 20 ms.

NEXT

' Continue forward for 75 pulses.

FOR pulseCount = 1 TO 75                    ' Loop sends 75 forward pulses.
  PULSOUT 13, 850                           ' 1.7 ms pulse to left servo.
  PULSOUT 12, 650                           ' 1.3 ms pulse to right servo.
  PAUSE 20                                  ' Pause for 20 ms.
NEXT
```

```
' Ramp down from going forward to a full stop.

FOR pulseCount = 100 TO 1                    ' Loop ramps down for 100 pulses.
  PULSOUT 13, 750 + pulseCount               ' Pulse = 1.5 ms + pulseCount.
  PULSOUT 12, 750 - pulseCount               ' Pulse = 1.5 ms - pulseCount.
  PAUSE 20                                   ' Pause for 20 ms.
NEXT

END                                          ' Stop until reset.
```


Your Turn

You can also create routines to combine ramping up or down with the 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 **pulseCount** is subtracted from 750 in the **PULSOUT 13** command, where before it was added. Likewise, **pulseCount** is added to the value of 750 in the **PULSOUT 12** command, where before it was subtracted.

```
' Ramp up to full speed going backwards

FOR pulseCount = 1 TO 100

  PULSOUT 13, 750 - pulseCount
  PULSOUT 12, 750 + pulseCount
  PAUSE 20

NEXT
```

You can also make a routine for ramping into a turn by adding the value of **pulseCount** to 750 in both **PULSOUT** commands. By subtracting **pulseCount** from 750 in both **PULSOUT** commands, you can ramp into a turn the other direction. Here's an example of a quarter turn with ramping. The servos don't get an opportunity to get up to full speed before they have to slow back down again.

```
' Ramp up right rotate.

FOR pulseCount = 0 TO 30

  PULSOUT 13, 750 + pulseCount
  PULSOUT 12, 750 + pulseCount
  PAUSE 20

NEXT
```

```
' Ramp down right rotate

FOR pulseCount = 30 TO 0

  PULSOUT 13, 750 + pulseCount
  PULSOUT 12, 750 + pulseCount
  PAUSE 20

NEXT
```

- ✓ Open ForwardLeftRightBackward.bs2 from Activity #1, and save it as ForwardLeftRightBackwardRamping.bs2.
- ✓ Modify the new program so your Boe-Bot will ramp into and out of each maneuver. Hint: you might use the code snippets above, and similar snippets from StartAndStopWithRamping.bs2.

ACTIVITY #5: SIMPLIFY NAVIGATION WITH SUBROUTINES

In the next chapter, your Boe-Bot will have to perform maneuvers to avoid obstacles. One of the key ingredients to avoiding obstacles is executing pre-programmed maneuvers. One way of executing pre-programmed maneuvers is with subroutines. This activity introduces subroutines, and also two different approaches to creating reusable maneuvers with subroutines.

Inside the Subroutine

There are two parts of a PBASIC subroutine. One part is the subroutine call. It's the command in the program that tells it to jump to the reusable part of code, then come back when it's done. The other part is the actual subroutine. It starts with a label that serves as its name and ends with a **`RETURN`** command. The commands between the label and the **`RETURN`** command make up the code block that does the job you want the subroutine to do.

Figure 4-5 shows part of a PBASIC program that contains a subroutine call and a subroutine. The subroutine call is the **`GOSUB My_Subroutine`** command. The actual subroutine is everything from the **`My_Subroutine:`** label through the **`RETURN`** command. Here's how it works. When the program gets to the **`GOSUB My_Subroutine`** command, it looks for the **`My_Subroutine:`** label. As shown by arrow (1), the program jumps to the **`My_Subroutine:`** label and starts executing commands. The program keeps going down line by line from the label, so you'll see the message "Command in subroutine" in

your Debug Terminal. **`PAUSE 1000`** causes a one second pause. Then, when the program gets to the **`RETURN`** command, arrow (2) shows how it jumps back to the command immediately after the **`GOSUB`** command. In this case, it's a **`DEBUG`** command that displays the message "After subroutine."

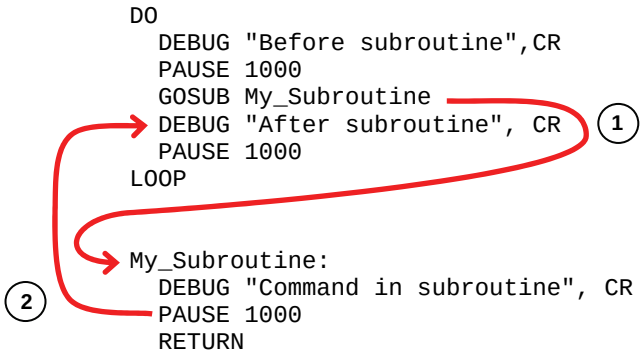


Figure 4-5
Subroutine Basics

Example Program – OneSubroutine.bs2

- ✓ Enter, save, and run OneSubroutine.bs2.

```
' Robotics with the Boe-Bot - OneSubroutine.bs2
' This program demonstrates a simple subroutine call.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Before subroutine",CR
PAUSE 1000
GOSUB My_Subroutine
DEBUG "After subroutine", CR
END

My_Subroutine:
  DEBUG "Command in subroutine", CR
  PAUSE 1000
  RETURN
```

- ✓ Watch your Debug Terminal, and press the Reset button a few times. You should get the same set of three messages in the right order each time.

Here's an example program that has two subroutines. One subroutine makes a high-pitched tone while the other makes a low-pitched tone. The commands between **DO** and **LOOP** call each of the subroutines in turn. Try this program and note the effect.

Example Program – TwoSubroutines.bs2

- ✓ Enter, save, and run TwoSubroutines.bs2.

```
' Robotics with the Boe-Bot - TwoSubroutines.bs2
' This program demonstrates that a subroutine is a reusable block of commands.

' {$STAMP BS2}
' {$PBASIC 2.5}

DO
  GOSUB High_Pitch
  DEBUG "Back in main", CR
  PAUSE 1000
  GOSUB Low_Pitch
  DEBUG "Back in main again", CR
  PAUSE 1000
  DEBUG "Repeat...",CR,CR
LOOP

High_Pitch:
  DEBUG "High pitch", CR
  FREQOUT 4, 2000, 3500
  RETURN

Low_Pitch:
  DEBUG "Low pitch", CR
  FREQOUT 4, 2000, 2000
  RETURN
```

Let's try putting the forward, left, right, and backward navigation routines inside subroutines. Here's an example:

Example Program – MovementsWithSubroutines.bs2

- ✓ Enter, save, and run MovementsWithSubroutines.bs2. Hint: you can use the Edit menu in the BASIC Stamp Editor to copy and paste code blocks from one program to another.

```
' Robotics with the Boe-Bot - MovementsWithSubroutines.bs2
' Make forward, left, right, and backward movements in reusable subroutines.
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

counter        VAR     Word

FREQOUT 4, 2000, 3000                        ' Signal program start/reset.

GOSUB Forward
GOSUB Left
GOSUB Right
GOSUB Backward

END

Forward:
  FOR counter = 1 TO 64
    PULSOUT 13, 850
    PULSOUT 12, 650
    PAUSE 20
  NEXT
  PAUSE 200
  RETURN

Left:
  FOR counter = 1 TO 24
    PULSOUT 13, 650
    PULSOUT 12, 650
    PAUSE 20
  NEXT
  PAUSE 200
  RETURN

Right:
  FOR counter = 1 TO 24
    PULSOUT 13, 850
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  PAUSE 200
  RETURN

Backward:
  FOR counter = 1 TO 64
    PULSOUT 13, 650
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN
```

You should recognize the pattern of movement your Boe-Bot makes; it is the same one made by ForwardLeftRightBackward.bs2. Clearly there are many different ways to structure a program that will result in the same movements. A third approach is given in the example below.

Example Program – MovementsWithVariablesAndOneSubroutine.bs2

Here's another example program that causes your Boe-Bot to perform the same maneuvers, but it only uses one subroutine and some variables to do it.

You have surely noticed that up to this point each Boe-Bot maneuver has been accomplished with similar code blocks. Compare these two snippets:

```
' Forward full speed
FOR counter = 1 TO 64
  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20
NEXT
```

```
' Ramp down from full speed backwards
FOR pulseCount = 100 TO 1
  PULSOUT 13, 750 - pulseCount
  PULSOUT 12, 750 + pulseCount
  PAUSE 20
NEXT
```

What causes these two code blocks to perform different maneuvers are changes to the **FOR** ***StartValue*** and ***EndValue*** arguments, and the **PULSOUT** ***Duration*** arguments. These arguments can be variables, and these variables can be changed repeatedly during program run time to generate different maneuvers. Instead of using separate subroutines with specific **PULSOUT** ***Duration*** arguments for each maneuver, the program below uses the same subroutine over and over. The key to making different maneuvers is to set the variables to the correct values for the maneuver you want before calling the subroutine.

- ✓ Enter, save, and run MovementWithVariablesAndOneSubroutine.bs2.

```
' Robotics with the Boe-Bot - MovementWithVariablesAndOneSubroutine.bs2
' Make a navigation routine that accepts parameters.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"
```

```
counter        VAR     Word
pulseLeft      VAR     Word
pulseRight     VAR     Word
pulseCount     VAR     Byte

FREQOUT 4, 2000, 3000                        ' Signal program start/reset.

' Forward
pulseLeft = 850: pulseRight = 650: pulseCount = 64: GOSUB Navigate

' Left turn
pulseLeft = 650: pulseRight = 650: pulseCount = 24: GOSUB Navigate

' Right turn
pulseLeft = 850: pulseRight = 850: pulseCount = 24: GOSUB Navigate

' Backward
pulseLeft = 650: pulseRight = 850: pulseCount = 64: GOSUB Navigate

END

Navigate:
  FOR counter = 1 TO pulseCount
    PULSOUT 13, pulseLeft
    PULSOUT 12, pulseRight
    PAUSE 20
  NEXT
  PAUSE 200
  RETURN
```


Did your Boe-Bot perform the familiar forward-left-right-backward sequence? This program may be difficult to read at first, because the instructions are arranged in a new way. Instead of having each variable statement and each **GOSUB** command on a different line, they are grouped together on the same line and separated by colons. Here, the colons function the same as a carriage return to separate each PBASIC instruction. Using colons this way allows all of the new variable values for a given maneuver to be stored together, and on the same line as the subroutine call.

Your Turn

Here is your "dead reckoning" contest mentioned earlier.

- ✓ Modify MovementWithVariablesAndOneSubroutine.bs2 to make your Boe-Bot drive in a square, facing forwards on the first two sides and backwards on the second two sides. Hint: you will need to use your own **PULSOUT** ***EndValue*** argument that you determined in Activity #2, page 109.

ACTIVITY #6: ADVANCED TOPIC—BUILDING COMPLEX MANEUVERS IN EEPROM

When you download PBASIC program to your BASIC Stamp, the BASIC Stamp Editor converts your program to numeric values called tokens. These tokens are what the BASIC Stamp uses as instructions for executing the program. They are stored in one of the two smaller black chips on top of your BASIC Stamp. This chip is a special type of computer memory called EEPROM, which stands for electrically erasable programmable read only memory (EEPROM). The BASIC Stamp's EEPROM can hold 2048 bytes (2 KB) of information. What's not used for program storage (which builds from address 2047 toward address 0) can be used for data storage (which builds from address 0 toward address 2047).

If the data you store in EEPROM collides with your program, the PBASIC program won't execute properly.

EEPROM memory is different from RAM (random access memory) variable storage in several respects:

- EEPROM takes more time to store a value, sometimes up to several milliseconds.
- EEPROM can accept a finite number of write cycles, around 10 million writes. RAM has unlimited read/write capabilities.
- The primary function of the EEPROM is to store programs; data can be stored in leftover space.

You can view the contents of the BASIC Stamp's EEPROM in the BASIC Stamp Editor by clicking Run and selecting Memory Map. Figure 4-6 shows the Memory Map for MovementsWithSubroutines.bs2. Note the condensed EEPROM Map on the left side of the figure. This shaded area in the small box at the bottom shows the amount of EEPROM that MovementsWithSubroutines.bs2 occupies.

The Memory Map images shown in this activity were taken from the BASIC Stamp Editor v2.1. If you are using a different version of the BASIC Stamp Editor, your memory map will contain the same information, but it may be formatted differently.

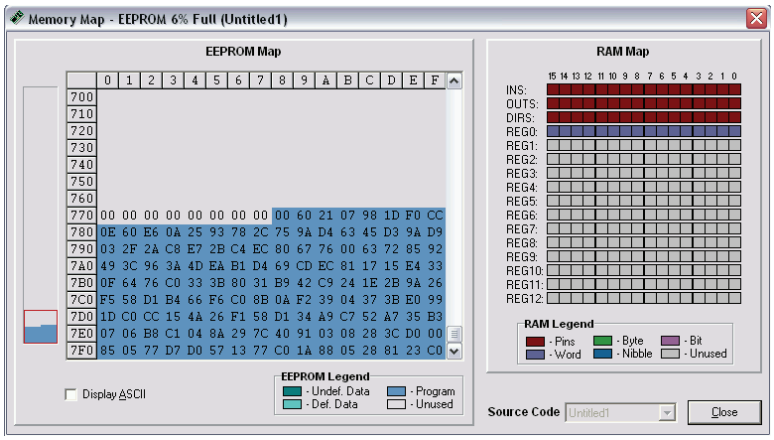


Figure 4-6
BASIC Stamp Editor Memory Map

4

While we are here, note also that the **`counter`** variable we declared as a word is visible in Register 0 of the RAM Map.

This program might have seemed large while you were typing it in, but it only takes up 136 of the available 2048 bytes of program memory. There currently is enough room for quite a long list of instructions. Since a character occupies a byte in memory, there is room for 1912 one-character direction instructions.

EEPROM Navigation

Up to this point we have tried three different programming approaches to make your Boe-Bot drive forward, turn left, turn right, and drive back again. Each technique has its merits, but all would be cumbersome if you wanted your Boe-Bot to execute a longer, more complex set of maneuvers. The upcoming program examples will use the now-familiar code blocks in subroutines for each basic maneuver. Each maneuver is given a one-letter code as a reference. Long lists of these code letters can be stored in EEPROM and then read and decoded during program execution. This avoids the tedium of repeating long lists of subroutines, or having to change the variables before each **`GOSUB`** command.

This programming approach requires some new PBASIC instructions: the **`DATA`** directive, and **`READ`** and **`SELECT...CASE...ENDSELECT`** commands. Let's take a look at each before trying out an example program.

Each of the basic maneuvers is given a single letter code that will correspond to its subroutine: F for **Forward**, B for **Backward**, L for **Left_Turn**, and R for **Right_Turn**. Complex Boe-Bot movements can be quickly choreographed by making a string of these code letters. The last letter in the string is a Q, which will mean "quit" when the movements are over. The list is saved in EEPROM during program download with the **DATA** directive, which looks like this:

```
DATA            "FLFFRBLBBQ"
```

Each letter is stored in a byte of EEPROM, beginning at address 0 (unless we tell it to start somewhere else). The **READ** command can then be used to get this list back out of EEPROM while the program is running. These values can be read from within a **DO…LOOP** like this:

```
DO UNTIL (instruction = "Q")
  READ address, instruction
  address = address + 1
  ' PBASIC code block omitted here.
LOOP
```

The **address** variable is the location of each byte in EEPROM that is holding a code letter. The **instruction** variable will hold the actual value of that byte, our code letter. Notice that each time through the loop, the value of the **address** variable is increased by one. This will allow each letter to be read from consecutive bytes in the EEPROM, starting at address 0.

The **DO…LOOP** command has optional conditions that are handy for different circumstances. The **DO UNTIL (*condition*)...LOOP** allows the loop to repeat until a certain condition occurs. **DO WHILE (*condition*)...LOOP** allows the loop to repeat only while a certain condition exists. Our example program will use **DO…LOOP UNTIL (*condition*).** In this case, it causes the **DO…LOOP** to keep repeating until the character "Q" is read from EEPROM.

A **SELECT...CASE...ENDSELECT** statement can be used to select a variable and evaluate it on a case-by-case basis and execute code blocks accordingly. Here is the code block that will look at each letter value held in the instruction variable and then call the appropriate subroutine for each instance, or case, of a given letter.

```
SELECT instruction
  CASE "F": GOSUB Forward
  CASE "B": GOSUB Backward
```

```
    CASE "R": GOSUB Right_Turn
    CASE "L": GOSUB Left_Turn
  ENDSELECT
```

Here are these concepts, all together in a single program.

Example Program: EepromNavigation.bs2

- ✓ Carefully read the code instructions and comments in EepromNavigation.bs2 to understand what each part of the program does.
- ✓ Enter, save, and run EepromNavigation.bs2.

```
' Robotics with the Boe-Bot - EepromNavigation.bs2
' Navigate using characters stored in EEPROM.
' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Variables ]-------------------------------------------------------

pulseCount     VAR     Word                ' Stores number of pulses.
address        VAR     Byte                ' Stores EEPROM address.
instruction    VAR     Byte                ' Stores EEPROM instruction.

' -----[ EEPROM Data ]-----------------------------------------------------

'      Address: 0123456789                 ' These two commented lines show
'               ||||||||||                 ' EEPROM address of each datum.
DATA           "FLFFRBLBBQ"                ' Navigation instructions.

' -----[ Initialization ]--------------------------------------------------

FREQOUT 4, 2000, 3000                      ' Signal program start/reset.

' -----[ Main Routine ]----------------------------------------------------

DO UNTIL (instruction = "Q")

  READ address, instruction                ' Data at address in instruction.
  address = address + 1                    ' Add 1 to address for next read.

  SELECT instruction                       ' Call a different subroutine
    CASE "F": GOSUB Forward                ' for each possible character
    CASE "B": GOSUB Backward               ' that can be fetched from
    CASE "L": GOSUB Left_Turn              ' EEPROM.
    CASE "R": GOSUB Right_Turn
  ENDSELECT

LOOP
```

```
END                                           ' Stop executing until reset.

' -----[ Subroutine - Forward ]-------------------------------------------

Forward:                                      ' Forward subroutine.
  FOR pulseCount = 1 TO 64                    ' Send 64 forward pulses.
    PULSOUT 13, 850                           ' 1.7 ms pulse to left servo.
    PULSOUT 12, 650                           ' 1.3 ms pulse to right servo.
    PAUSE 20                                  ' Pause for 20 ms.
  NEXT
  RETURN                                      ' Return to Main Routine loop.

' -----[ Subroutine - Backward ]------------------------------------------

Backward:                                     ' Backward subroutine.
  FOR pulseCount = 1 TO 64                    ' Send 64 backward pulses.
    PULSOUT 13, 650                           ' 1.3 ms pulse to left servo.
    PULSOUT 12, 850                           ' 1.7 ms pulse to right servo.
    PAUSE 20                                  ' Pause for 20 ms.
  NEXT
  RETURN                                      ' Return to Main Routine loop.

' -----[ Subroutine - Left_Turn ]------------------------------------------

Left_Turn:                                    ' Left turn subroutine.
  FOR pulseCount = 1 TO 24                    ' Send 24 left rotate pulses.
    PULSOUT 13, 650                           ' 1.3 ms pulse to left servo.
    PULSOUT 12, 650                           ' 1.3 ms pulse to right servo.
    PAUSE 20                                  ' Pause for 20 ms.
  NEXT
  RETURN                                      ' Return to Main Routine loop.

' -----[ Subroutine – Right_Turn ]-----------------------------------------

Right_Turn:                                   ' right turn subroutine.
  FOR pulseCount = 1 TO 24                    ' Send 24 right rotate pulses.
    PULSOUT 13, 850                           ' 1.7 ms pulse to left servo.
    PULSOUT 12, 850                           ' 1.7 ms pulse to right servo.
    PAUSE 20                                  ' Pause for 20 ms.
  NEXT
  RETURN                                      ' Return to Main Routine section.
```

Did your Boe-Bot drive in a rectangle, going forward on the first two sides and backwards on the second two? If it looked more like a trapezoid, you may want to adjust the **`FOR...NEXT`** loop's ***EndValue*** arguments in the turning subroutines to make precise 90-degree turns.

Your Turn

- ✓ With EepromNavigation.bs2 active in the BASIC Stamp Editor, click Run and select Memory Map.

Your stored instructions will appear highlighted in blue at the beginning of the Detailed EEPROM Map as shown in Figure 4-7. The numbers shown are the hexadecimal ASCII codes that correspond to the characters you entered in your **DATA** statement.

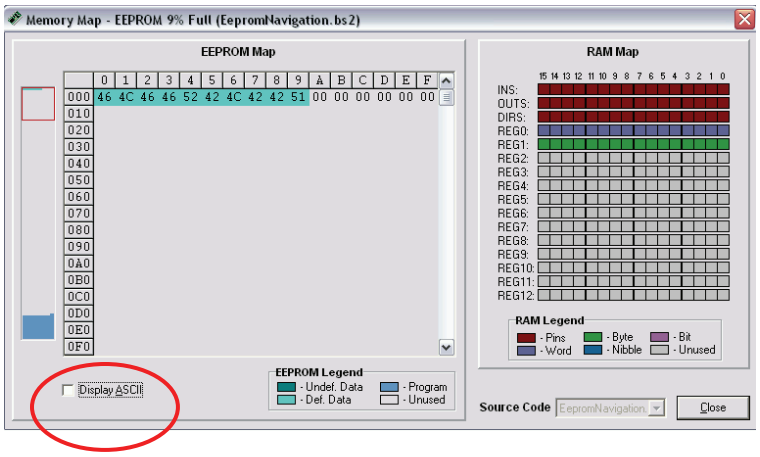


Figure 4-7
Memory Map with Stored Instructions Visible in EEPROM Map

- ✓ Click the Display ASCII checkbox near the lower left corner of the Memory Map window.

Now the direction instructions will appear in a more familiar format shown in Figure 4-8. Instead of ASCII codes, they appear as the actual characters you recorded using the **DATA** directive.

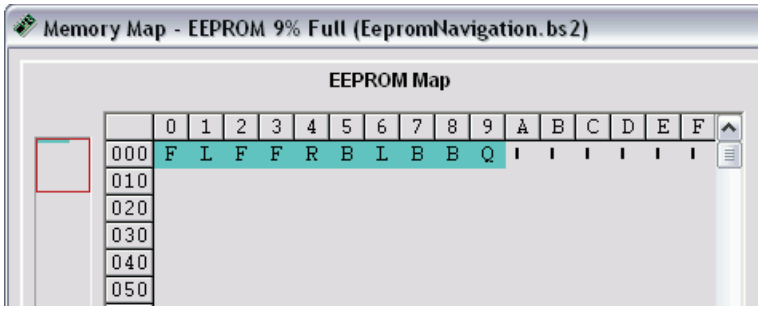


Figure 4-8
Close-up of the Detailed EEPROM Map after Display ASCII Box is Checked

This program stored a total of 10 characters in EEPROM. These ten characters were accessed by the **READ** command's **address** variable. The **address** variable was declared as a byte, so it can access up to 256 locations, well over the 10 we needed. If the **address** variable is re-declared to be a word variable, you could theoretically access up to 65535, far more locations than are available. Keep in mind that if your program gets larger, the number of available EEPROM addresses for holding data gets smaller.

You can modify the existing data string to a new set of directions. You can also add additional **DATA** statements. The data is stored sequentially, so the first character in the second data string will get stored immediately after the last character in the first data string.

- ✓ Try changing, adding, and deleting characters in the **DATA** directive, and re-running the program. Remember that the last character in the **DATA** directive should always be a "Q."
- ✓ Modify the **DATA** directive to make your Boe-Bot perform the familiar forward-left-right-backward sequence of movements.
- ✓ Try adding a second **DATA** directive. Remember to remove the "Q" from the end of the first **DATA** directive and add it to the end of the second. Otherwise, the program will execute only the commands in the first **DATA** directive.

Example Program – EepromNavigationWithWordValues.bs2

This next example program looks complicated at first, but it is a very efficient way to design programs for custom Boe-Bot choreography. This example program uses EEPROM data storage, but does not use subroutines. Instead, a single code block is used, with variables in place of the **FOR...NEXT** loop's ***EndValue*** and **PULSOUT** ***Duration*** arguments.

By default, the **DATA** directive stores bytes of information in EEPROM. To store word-sized data items, you can add the **Word** modifier to the **DATA** directive, before each data item in your string. Each word-sized data item will use two bytes of EEPROM storage, so the data will be accessed via every other address location. When using more than one **DATA** directive, it is most convenient to assign a label to each one. This way, your **READ** commands can refer to the label to retrieve data items without you having to figure out at which EEPROM address each string of data items begins. Take a look at this code snippet:

```
' addressOffset   0         2         4         6        8
Pulses_Count DATA Word 64,  Word 24,  Word 24,  Word 64, Word 0
Pulses_Left  DATA Word 850, Word 650, Word 850, Word 650
Pulses_Right DATA Word 650, Word 650, Word 850, Word 850
```

Each of the three **DATA** statements begins with its own label. The **Word** modifier goes before each data item, and the items are separated by commas. These three strings of data will be stored in EEPROM one after another. We don't have to do the math to figure out the address number of a given data item, because the labels and the **addressOffset** variable will do that automatically. The **READ** command uses each label to determine the EEPROM address where that string begins, and then adds the value of the **addressOffset** variable to know how many address numbers to shift over to find the correct ***DataItem***. The ***DataItem*** found at the resulting ***Address*** will be stored in the **READ** command's ***Variable*** argument. Notice that the **Word** modifier also comes before the variable that stores the value fetched from EEPROM.

```
DO
  READ Pulses_Count + addressOffset, Word pulseCount
  READ Pulses_Left + addressOffset, Word pulseLeft
  READ Pulses_Right + addressOffset, Word pulseRight

  addressOffset = addressOffset + 2

  ' PBASIC code block omitted here.
LOOP UNTIL (pulseCount = 0)
```

The first time through the loop, **addressOffset** = 0. The first **READ** command will retrieve a value of 64 from the first address at the **Pulses_Count** label, and place it in the **pulseCount** variable. The second **READ** command retrieves a value of 850 from the first address specified by the **Pulses_Left** label, and places it in the **pulseLeft** variable. The third **READ** command retrieves a value of 650 from the first address specified by the **Pulses_Right** label and places it in the **pulseRight** variable. Notice that these are the three values in the "0" column of the code snippet above. When the value of those variables are placed in the code block that follows, this:

```
FOR counter = 1 TO pulseCount
  PULSOUT 13, pulseLeft
  PULSOUT 12, pulseRight
  PAUSE 20
NEXT
```

....becomes....

```
FOR counter = 1 TO 64
  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20
NEXT
```

Do you recognize the basic maneuver generated by this code block?

- ✓ Look at the other columns of the code snippet on page 133 and anticipate what the **FOR…NEXT** code block will look like on the second, third, and fourth times through the loop.
- ✓ Look at the **LOOP UNTIL (pulseCount = 0)** statement in the program below. The **<>** operator stands for "not equal to." What will happen on the fifth time through the loop?
- ✓ Enter, save, and run EepromNavigationWithWordValues.bs2.

```
' Robotics with the Boe-Bot - EepromNavigationWithWordValues.bs2
' Store lists of word values that dictate.

' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Variables ]-------------------------------------------------------

counter        VAR     Word
pulseCount     VAR     Word                ' Stores number of pulses.
addressOffset  VAR     Byte                ' Stores offset from label.
instruction    VAR     Byte                ' Stores EEPROM instruction.
pulseRight     VAR     Word                ' Stores servo pulse widths.
pulseLeft      VAR     Word

' -----[ EEPROM Data ]-----------------------------------------------------

' addressOffset       0         2         4         6        8
Pulses_Count DATA     Word 64,  Word 24,  Word 24,  Word 64, Word 0
Pulses_Left  DATA     Word 850, Word 650, Word 850, Word 650
Pulses_Right DATA     Word 650, Word 650, Word 850, Word 850

' -----[ Initialization ]--------------------------------------------------

FREQOUT 4, 2000, 3000                       ' Signal program start/reset.

' -----[ Main Routine ]----------------------------------------------------

DO

  READ Pulses_Count + addressOffset, Word pulseCount
  READ Pulses_Left + addressOffset, Word pulseLeft
  READ Pulses_Right + addressOffset, Word pulseRight

  addressOffset = addressOffset + 2
```

```
  FOR counter = 1 TO pulseCount
    PULSOUT 13, pulseLeft
    PULSOUT 12, pulseRight
    PAUSE 20
  NEXT


LOOP UNTIL (pulseCount = 0)

END                                          ' Stop executing until reset.
```


Did your Boe-Bot perform the familiar forward-left-right-backwards movements? Are you thoroughly bored with it by now? Do you want to see your Boe-Bot do something else, or to choreograph your own routine?

Your Turn – Making Your Own Custom Navigation Routines

- ✓ Save EepromNavigationWithWordValues.bs2. under a new name.
- ✓ Replace the **DATA** directives with the ones below.
- ✓ Run the modified program and see what your Boe-Bot does.

```
Pulses_Count DATA Word 60,  Word 80,  Word 100, Word 110,
                  Word 110, Word 100, Word 80,  Word 60,  Word 0
Pulses_Left  DATA Word 850, Word 800, Word 785, Word 760, Word 750,
                  Word 740, Word 715, Word 700, Word 650, Word 750
Pulses_Right DATA Word 650, Word 700, Word 715, Word 740, Word 750,
                  Word 760, Word 785, Word 800, Word 850, Word 750
```

- ✓ Make a table with three rows, one for each **DATA** directive, and a column for each Boe-Bot maneuver you want to make, plus one for the **Word 0** item in the **Pulses_Count** row.
- ✓ Use the table to plan out your Boe-Bot choreography, filling in the **FOR...NEXT** loop's ***EndValue*** and **PULSOUT** ***Duration*** arguments you will need for each maneuver's code block.
- ✓ Modify your program with your newly charted **DATA** directives.
- ✓ Enter, save, and run your custom program. Did your Boe-Bot do what you wanted it to do? Keep working on it until it does.

SUMMARY

This chapter introduced the basic Boe-Bot maneuvers: forward, backward, rotating in place to turn to the right or left, and pivoting. The type of maneuver is determined by the **PULSOUT** commands' ***Duration*** arguments. How far the maneuver goes is determined by the **FOR…NEXT** loop's ***StartValue*** and ***EndValue*** arguments.

Chapter 2 included a hardware adjustment, physically centering the Boe-Bot's servos with a screwdriver. This chapter focused on fine tuning adjustments made by manipulating the software. Specifically, a difference in rotation speed between the two servos was compensated for by changing the **PULSOUT** command's ***Duration*** argument for the faster of the two servos. This changes the Boe-Bot's path from a curve to a straight line if the servos are not perfectly matched. To refine turning so that the Boe-Bot turns to the desired angle, the ***StartValue*** and ***EndValue*** arguments of a **FOR…NEXT** loop can be adjusted.

Programming the Boe-Bot to travel a pre-defined distance can be accomplished by measuring the distance it travels in one second, with the help of a ruler. Using this distance, and the number of pulses in one second of run time, you can calculate the number of pulses required to cover a desired distance.

Ramping was introduced as a way to gradually accelerate and decelerate. It's kinder to the servos, and we recommended that you use your own ramping routines in place of the abrupt start and stop routines shown in the example programs. Ramping is accomplished by taking the same variable that's used as the ***Counter*** argument in a **FOR…NEXT** loop and adding it to or subtracting it from 750 in the **PULSOUT** command's ***Duration*** argument.

Subroutines were introduced as a way to make pre-programmed maneuvers reusable by a PBASIC program. Instead of writing an entire **FOR…NEXT** loop for each new maneuver, a single subroutine that contains a **FOR…NEXT** loop can be executed as needed with the **GOSUB** command. A subroutine begins with a label, and ends with the **RETURN** command. A subroutine is called from the main program with a **GOSUB** command. When the subroutine is finished and it encounters the **RETURN** command, the next command to be executed is the one immediately following the **GOSUB** command.

The BASIC Stamp's EEPROM stores the program it runs, but you can take advantage of any unused portion of the program to store values. This is a great way to store custom navigation routines. The **DATA** directive can store values in EEPROM. Bytes are stored

by default, but adding the **Word** modifier to each data item allows you to store values up to 65535 in two bytes' worth of EEPROM memory space. You can read values back out of EEPROM using the **READ** command. If you are retrieving a word-sized variable, make sure to place a **Word** modifier before the variable that will receive the value that **READ** fetches. **SELECT…CASE** was introduced as a way of evaluating a variable on a case by case basis, and executing a different code block depending on the case. Optional **DO…LOOP** conditions are helpful in certain circumstances; **DO UNTIL** ***(Condition)*****...LOOP** and **DO...LOOP UNTIL** ***(Condition)*** were demonstrated as ways to keep executing a **DO…LOOP** until a particular condition is detected.

Questions

1. What direction does the left wheel have to turn to make the Boe-Bot go forward? What direction does the right wheel have to turn?
2. When the Boe-Bot pivots to the left, what are the right and left wheels doing? What PBASIC commands do you need to make the Boe-Bot pivot left?
3. If your Boe-Bot veers slightly to the left when you are running a program 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-Bot travels 11 in/s, how many pulses will it take to make it travel 36 inches?
5. What's the relationship between a **FOR…NEXT** loop's ***Counter*** argument and the **PULSOUT** command's ***Duration*** argument that makes ramping possible?
6. What directive can you use to pre-store values in the BASIC Stamp's EEPROM before running a program?
7. What command can you use to retrieve a value stored in EEPROM and copy it to a variable?
8. What code block 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 are the different conditions that can be used with **DO…LOOP**?

Exercises

1. Write a routine that makes the Boe-Bot back up for 350 pulses.
2. Let's say that you tested your servos and discovered that it takes 48 pulses to make a 180° turn with right-rotate. With this information, write routines to make the Boe-Bot perform 30, 45, and 60 degree turns.
3. Write a routine that makes the Boe-Bot go straight forward, then ramp in and out of a pivoting turn, and then continue straight forward.

Projects

1. It is time to fill in column 3 of Table 2-1 on page 63. To do this, modify the **PULSOUT** ***Duration*** arguments in the program BoeBotForwardThreeSeconds.bs2 using each pair of values from column 1. Record your Boe-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-Bot maneuvers.
2. Figure 4-9 shows two simple courses. Write a program that will make your Boe-Bot navigate along each figure. Assume straight line distances (including the diameter of the circle) to be either 1 yd or 1 m.

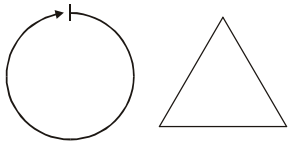


Figure 4-9
Simple Courses

Solutions

Q1. Left wheel counterclockwise, right wheel clockwise.
Q2. The right wheel is turning clockwise (forward), and the left wheel is not moving.

```
PULSOUT 13, 750
PULSOUT 12, 650
```

Q3. You can slow down the right wheel to correct a veer to the left. The **PULSOUT** command for the right wheel needs to be adjusted.

```
PULSOUT 12, 650
```

Adjust the 650 to something closer to 750 to slow the wheel down.

```
PULSOUT 12, 663
```

Q4. Given the data below, it should take 133 pulses to travel 36 inches:
Boe-Bot speed = 11 in/s
Boe-Bot distance = 36 in/s
pulses = (Boe-Bot distance / Boe-Bot speed) * (40.65 pulses / s)
= (36 / 11) * (40.65)
= 133.04
= 133

Q5. The **FOR…NEXT** loop's **pulseCount** variable can be used as an offset (plus or minus) to 750 (the center position) in the ***Duration*** argument.

```
FOR pulseCount = 1 to 100
  PULSOUT 13, 750 + pulseCount
  PULSOUT 12, 750 – pulseCount
  PAUSE 20
NEXT
```


Q6. The **DATA** directive.
Q7. The **READ** command
Q8. **SELECT...CASE...ENDSELECT**.
Q9. **UNTIL** and **WHILE**.

E1.
```
FOR counter = 1 to 350      ' Backward
  PULSOUT 13, 650
  PULSOUT 12, 850
  PAUSE 20
NEXT
```

E2.
```
FOR counter = 1 to 8       ' Rotate right 30 degrees
  PULSOUT 13, 850
  PULSOUT 12, 850
  PAUSE 20
NEXT

FOR counter = 1 to 12      ' Rotate right 45 degrees
  PULSOUT 13, 850
  PULSOUT 12, 850
  PAUSE 20
NEXT

FOR counter = 1 to 16      ' Rotate right 60 degrees
  PULSOUT 13, 850
  PULSOUT 12, 850
  PAUSE 20
NEXT
```

E3.
```
FOR counter = 1 to 100     ' Forward
  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20
NEXT
```

```
FOR counter = 0 TO 30      ' Ramping pivot turn
  PULSOUT 13, 750 + counter
  PULSOUT 12, 750
  PAUSE 20
NEXT

FOR counter = 30 TO 0
  PULSOUT 13, 750 + counter
  PULSOUT 12, 750
  PAUSE 20
NEXT

FOR counter = 1 to 100           ' Forward
  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20
NEXT
```

P1.

P13	P12	Description	Behavior
850	650	Full Speed: P13 CCW, P12 CW	Forward
650	850	Full Speed: P13 CW, P12 CCW	Backward
850	850	Full Speed: P13 CCW, P12 CCW	Right rotate
650	650	Full Speed: P13 CW, P12 CW	Left rotate
750	850	P13 Stopped, P12 CCW Full speed	Pivot back left
650	750	P13 CW Full Speed, P12 Stopped	Pivot back right
750	750	P13 Stopped, P12 Stopped	Stopped
760	740	P13 CCW Slow, P12 CW Slow	Forward slow
770	730	P13 CCW Med, P12 CW Med	Forward medium
850	700	P13 CCW Full Speed, P12 CW Medium	Veer right
800	650	P13 CCW Medium, P12 CW Full Speed	Veer left

P2. The circle can be implemented by veering right continuously. Trial and error, a yard or meter stick, will help you arrive at the right **PULSOUT** value. Circle with a one-yard diameter:

```
' Robotics with the Boe-Bot - Chapter 4 - Circle.bs2
' Boe-Bot navigates a circle of  1 yard diameter.

'{$STAMP BS2}
'{$PBASIC 2.5}
DEBUG "Program running!"
```

```
pulseCount     VAR     Word          ' Pulse count to servos

FREQOUT 4, 2000, 3000                ' Signal program start/reset.

' -----[ Main Routine ]-----------------------------------------------
Main:
DO
  PULSOUT 13, 850                    ' Veer right
  PULSOUT 12, 716
  PAUSE 20
LOOP
```


To make the triangle, first calculate the number of pulses required for a one meter or yard straight line, as in Question 4. Then fine-tune your distances to match your Boe-Bot and particular surface. For a triangle pattern, the Boe-Bot must travel 1 meter/yard forward, and then make a 120 degree turn. This should be repeated three times for the three sides of the triangle. You may have to adjust the **`pulseCount`** ***EndValue*** in the **`Right_Rotate120`** subroutine to get a precise 120 degree turn.

```
' Robotics with the Boe-Bot - Chapter 4 - Triangle.bs2
' Boe-Bot navigates triangle shape with 1 yard sides.
' Go forward, then turn 120 degrees.  Repeat three times.

'{$STAMP BS2}
'{$PBASIC 2.5}
DEBUG "Program running!"

counter        VAR     Nib           ' Triangle has 3 sides
pulseCount     VAR     Word          ' Pulse count to servos

FREQOUT 4, 2000, 3000                ' Signal program start/reset.

Main:
  FOR counter = 1 TO 3               ' Repeat 3 times for triangle
    GOSUB Forward
    GOSUB Right_Rotate120
  NEXT
  END

Forward:
  FOR pulseCount = 1 TO 163          ' Forward 1 yard
    PULSOUT 13, 850
    PULSOUT 12, 650
    PAUSE 20
  NEXT
  RETURN
```

```
Right_Rotate120:
  FOR pulseCount = 1 TO 21           ' Rotate right 120 degrees
    PULSOUT 13, 850
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN
```

Chapter 5: Tactile Navigation with Whiskers

Many types of robotic machinery rely on a variety of tactile switches. For example, a tactile switch may detect when a robotic arm has encountered an object. The robot can be programmed to pick up the object and place it elsewhere. Factories use tactile switches to count objects on a production line, and also for aligning objects during industrial processes. In all these instances, the switches provide inputs that dictate some other form of programmed output. The inputs are electronically monitored by the product, be it a robot, or a calculator, or a production line. Based on the state of the switches, the robot arm grabs an object, or the calculator display updates, or the factory production line reacts with motors or servos to guide products.

In this chapter, you will build tactile switches, called whiskers, onto your Boe-Bot and test them. You will then program the Boe-Bot to monitor the state 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

The whiskers are so named because that is what these bumper switches look like, though some argue they look more like antennae. At any rate, these whiskers are shown mounted on a Boe-Bot in Figure 5-1. Whiskers give the Boe-Bot the ability to sense the world around it through touch, much like the antennae on an ant or the whiskers on a cat. 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 to increase your Boe-Bot's functionality.

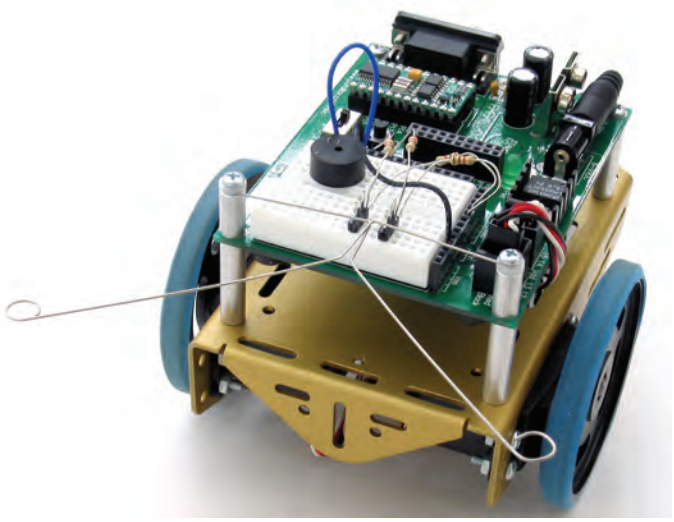


Figure 5-1
Boe-Bot with Whiskers

ACTIVITY #1: BUILDING AND TESTING THE WHISKERS

Before moving on to programs that make the Boe-Bot navigate based on what it can touch, it's essential to build and test the whiskers first. This activity will guide you through building and testing the whiskers.

Whisker Circuit and Assembly

- ✓ Gather the whiskers hardware shown in Figure 5-2.
- ✓ 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)

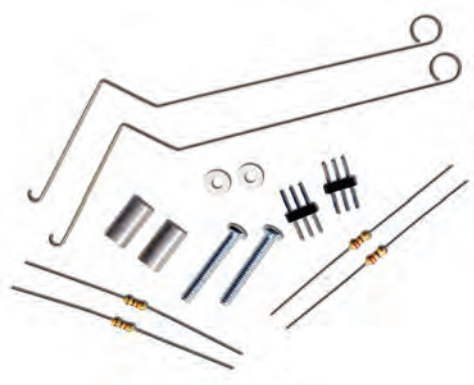


Figure 5-2
Whiskers Hardware

Building the Whiskers

- ✓ Remove the two front screws that hold your board to the front standoffs.
- ✓ Refer to Figure 5-3 while following the remaining instructions.
- ✓ 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.

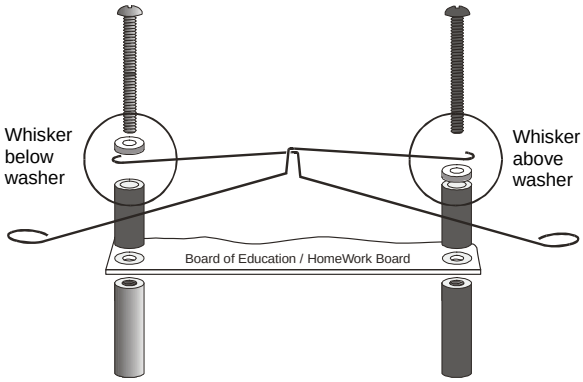


Figure 5-3
Mounting the Whiskers

The next step is add the whiskers circuit shown in Figure 5-4 to the piezospeaker and servo circuits you built and tested in Chapter 2 and Chapter 3.

- ✓ If you have a Board of Education, build the whiskers circuit shown in Figure 5-4 using the wiring diagram in Figure 5-5 on page 147 as a reference.
- ✓ If you have a HomeWork Board, build the whiskers circuit shown in Figure 5-4 using the wiring diagram in Figure 5-6 on page 148 as a reference.
- ✓ 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 a recommended starting point.

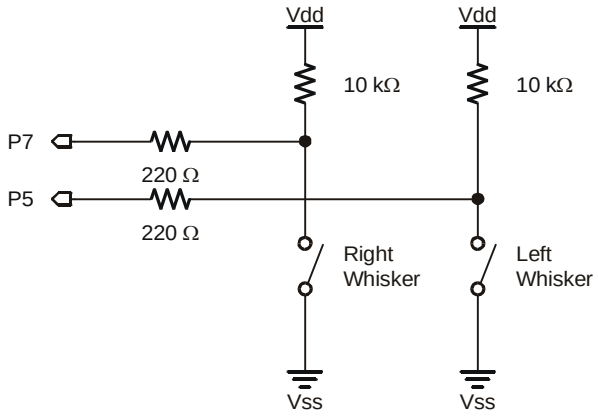
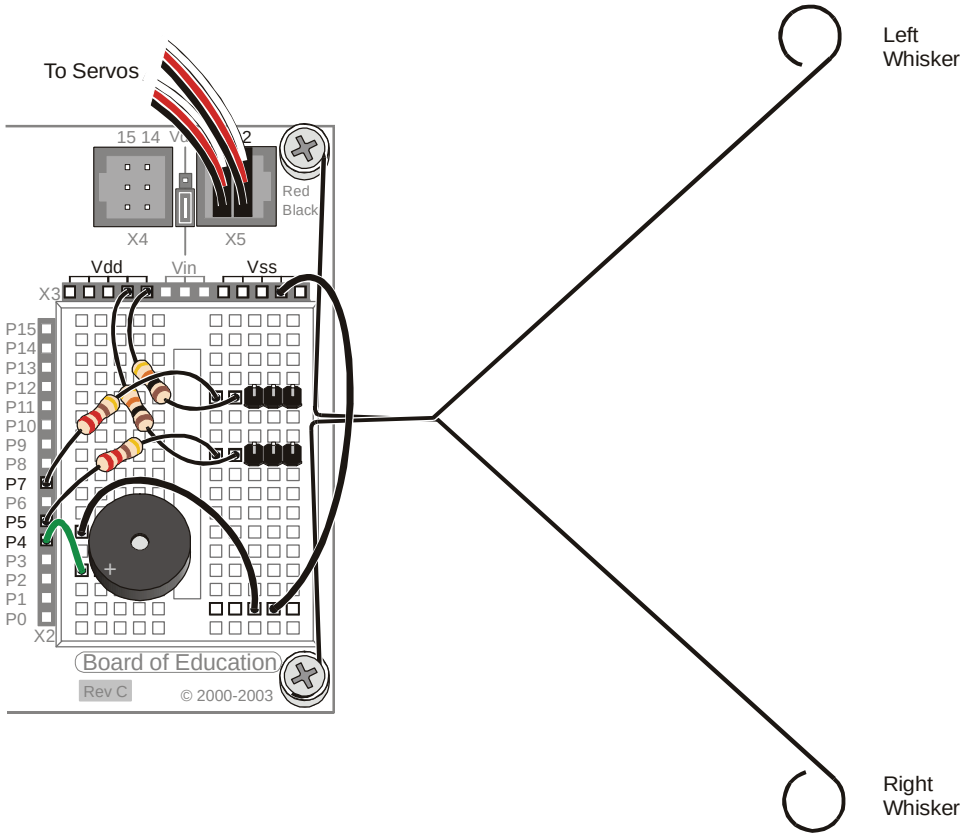


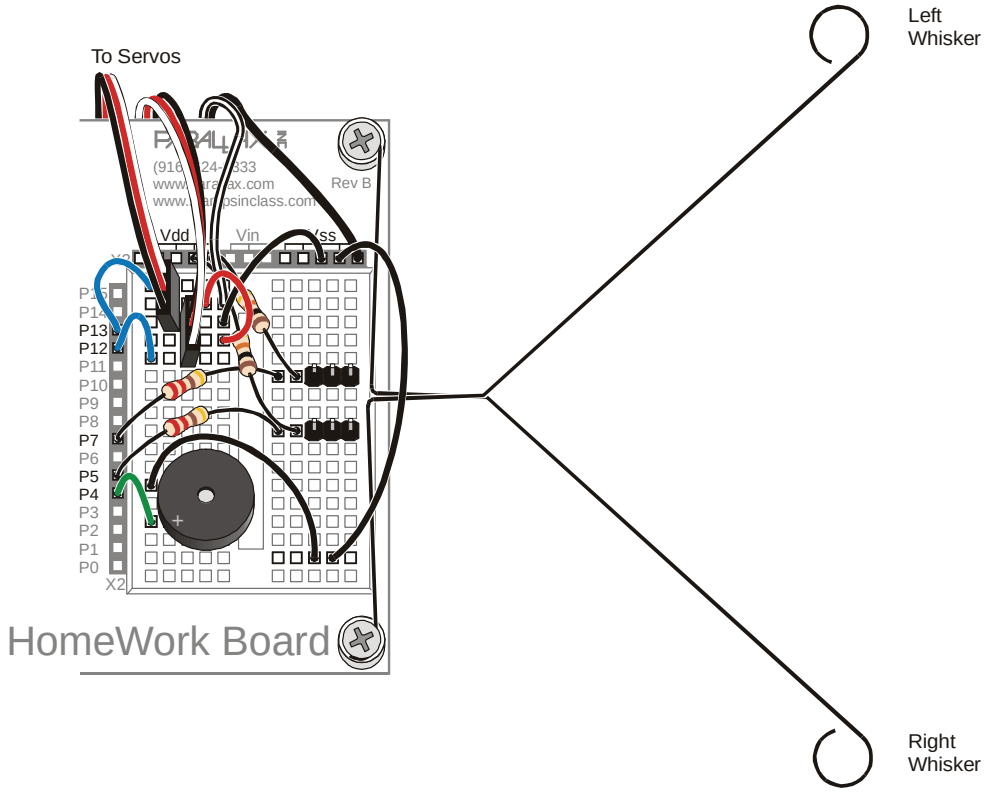
Figure 5-4
Whiskers Schematic

Figure 5-5: Whisker Wiring Diagram for the Board of Education



Use the 220 Ω resistors (red-red-brown color codes) to connect P5 and P7 to their corresponding 3-pin headers. Use the 10 kΩ resistors (brown-black-orange color codes) to connect Vdd to each 3-pin header.

Figure 5-6: Whisker Wiring Diagram for the HomeWork Board



Use the 220 Ω resistors (red-red-brown color codes) to connect P5 and P7 to their corresponding 3-pin headers. Use the 10 kΩ resistors (brown-black-orange color codes) to connect Vdd to each 3-pin header.

Testing the Whiskers

Take a second look at the whiskers schematic (Figure 5-7). Each whisker is both the mechanical extension and the ground electrical connection of a normally open, single-pole, single-throw switch. The reason the whiskers are connected to ground (Vss) is because the plated holes at the outer edge of the board are all connected to Vss. This is true for both the Board of Education and the BASIC Stamp HomeWork Board. The metal standoffs and screw provide the electrical connection to each whisker.

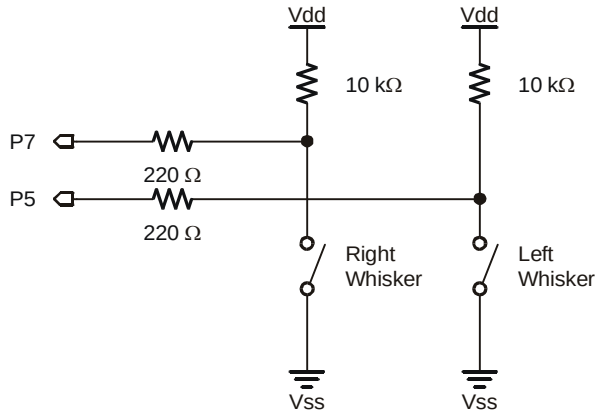


Figure 5-7
Whiskers Schematic
A Second Look

The BASIC Stamp can be programmed to detect when a whisker is pressed. I/O pins connected to each switch circuit monitor the voltage at the 10 kΩ pull-up resistor. Figure 5-8 illustrates how this works. When a given whisker is not pressed, the voltage at the I/O pin connected to that whisker is 5 V. When a whisker is pressed, the I/O line is shorted to ground (Vss), so the I/O line sees 0 V.

All I/O pins default to input every time a PBASIC program starts. This means that the I/O pins connected to the whiskers will function as inputs automatically. As an input, an I/O pin connected to a whisker circuit will cause its input register to store a 1 if the voltage is 5 V (whisker not pressed) or a 0 if the voltage is 0 V (whisker pressed). The Debug Terminal can be used to display these values.

How do you get the BASIC Stamp to tell you whether it's reading a 1 or 0?

Because the circuit is connected to P7, this 1 or 0 value will appear in a variable named **IN7**. **IN7** is called an input register. Input register variables are built-in and do not have to be declared in the beginning of your program. You can see the value this variable is storing by using the command **DEBUG BIN1 IN7**. The **BIN1** is a formatter that tells the Debug Terminal to display one binary digit (either 1 or 0).

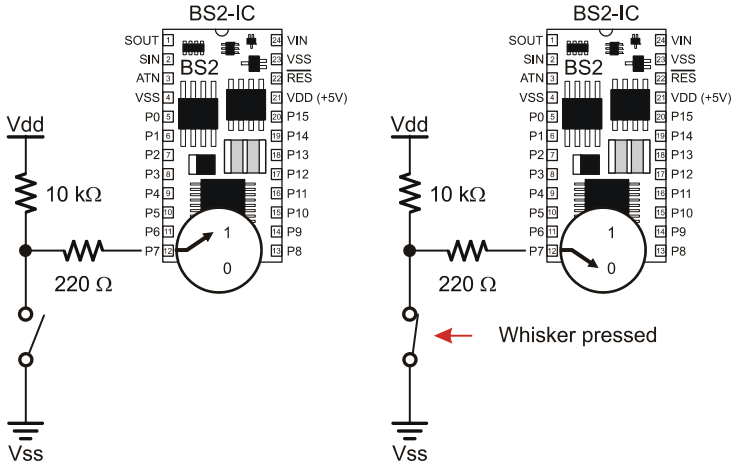


Figure 5-8
Detecting Electrical Contacts

Example Program: TestWhiskers.bs2

This next example program is designed to test the whiskers to make sure they are functioning properly. By displaying the binary digits stored in the P7 and P5 input registers (**IN7** and **IN5**), the program will show you whether the BASIC Stamp detects contact with a whisker. When the value stored in a given input register is 1, the whisker is not pressed. When it is 0, the whisker is pressed.

- ✓ Reconnect power to your board and servos.
- ✓ Enter, save, and run TestWhiskers.bs2.
- ✓ This program makes use of the Debug Terminal, so leave the programming cable connected to the BASIC Stamp while the program is running.

```
' Robotics with the Boe-Bot - TestWhiskers.bs2
' Display what the I/O pins connected to the whiskers sense.
' {$STAMP BS2}                                ' Stamp directive.
' {$PBASIC 2.5}                               ' PBASIC directive.
```

```
DEBUG "WHISKER STATES", CR,
      "Left     Right", CR,
      "------   ------"

DO
  DEBUG CRSRXY, 0, 3,
        "P5 = ", BIN1 IN5,
        "   P7 = ", BIN1 IN7
  PAUSE 50
LOOP
```


- ✓ Note the values displayed in the Debug Terminal; it should display that both P7 and P5 are equal to 1.
- ✓ Check Figure 5-5 on page 147 (or Figure 5-6 on page 148) so you know which r is the "left whisker" and which is the "right whisker."
- ✓ Press the right whisker into its three-pin header, and note the values displayed in the Debug Terminal. It should now read:
  P5 = 1 P7 = 0
- ✓ Press the left whisker into its three-pin header, and note the value displayed in the Debug Terminal again. This time it should read:
  P5 = 0 P7 = 1
- ✓ Press both whiskers against both three-pin headers. Now it should read
  P5 = 0 P7 = 0
- ✓ If the whiskers passed all these tests, you're ready to move on; otherwise, check your program and circuits for errors.

What is a Cursor? What is CRSRXY?

According to Merriam-Webster online dictionary, a cursor is: "A moveable item used to mark a position as…a visual cue on a video display that indicates position." As you move your mouse, the pointer that moves on your screen is a cursor. The Debug Terminal's cursor is somewhat different because it doesn't flash or do anything to indicate its position. But, wherever the Debug Terminal's cursor is, that's where the next character gets printed.

It is a formatter that allows you to conveniently arrange information your program sends to the Debug Terminal. The formatter **`CRSRXY 0, 3,`** in the command:

```
DEBUG CRSRXY, 0, 3,
        "P5 = ", BIN1 IN5,
        "   P7 = ", BIN1 IN7
```

...places the cursor at column 0, row 3 in the Debug Terminal. This makes it display nicely below the "Whisker States" table heading. Each time through the loop, the new values overwrite the old values because the cursor keeps going back to the same place.

ACTIVITY #2: FIELD TESTING THE WHISKERS

Assume that you may have to test the whiskers at some later time away from a computer. Since the Debug Terminal won't be available, what can you do? One solution would be to program the BASIC Stamp so that it sends an output signal that corresponds to the input signal it's receiving. This can be done with a pair of LED circuits and a program that turns the LEDs on and off based on the whisker inputs.

Parts List:

(2) Resistors, 220 Ω (red-red-brown)
(2) LEDs, red

Building the LED Whisker Testing Circuits

- ✓ Disconnect power from your board and servos.
- ✓ If you have a Board of Education, add the circuit shown in Figure 5-9 with the help of the wiring diagram in Figure 5-10 (page 153).
- ✓ If you have a HomeWork Board, add the circuit shown in Figure 5-9 with the help of the wiring diagram in Figure 5-11 (page 154).

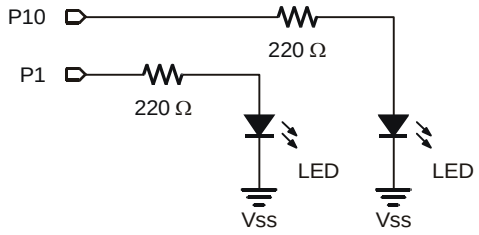


Figure 5-9
LED Whisker Testing Schematic

Add these LED circuits.

Remember that an LED is a one way current valve.

If it is plugged in backwards, it will not let current pass through, and so will not emit light. For the LED to emit light when the BASIC Stamp sends a high signal, the LED's anode must be connected to the 220 Ω resistor, and its cathode must be connected to Vss. See Figure 5-10 or Figure 5-11.

Figure 5-10 Whisker Plus LED Wiring Diagram for the Board of Education

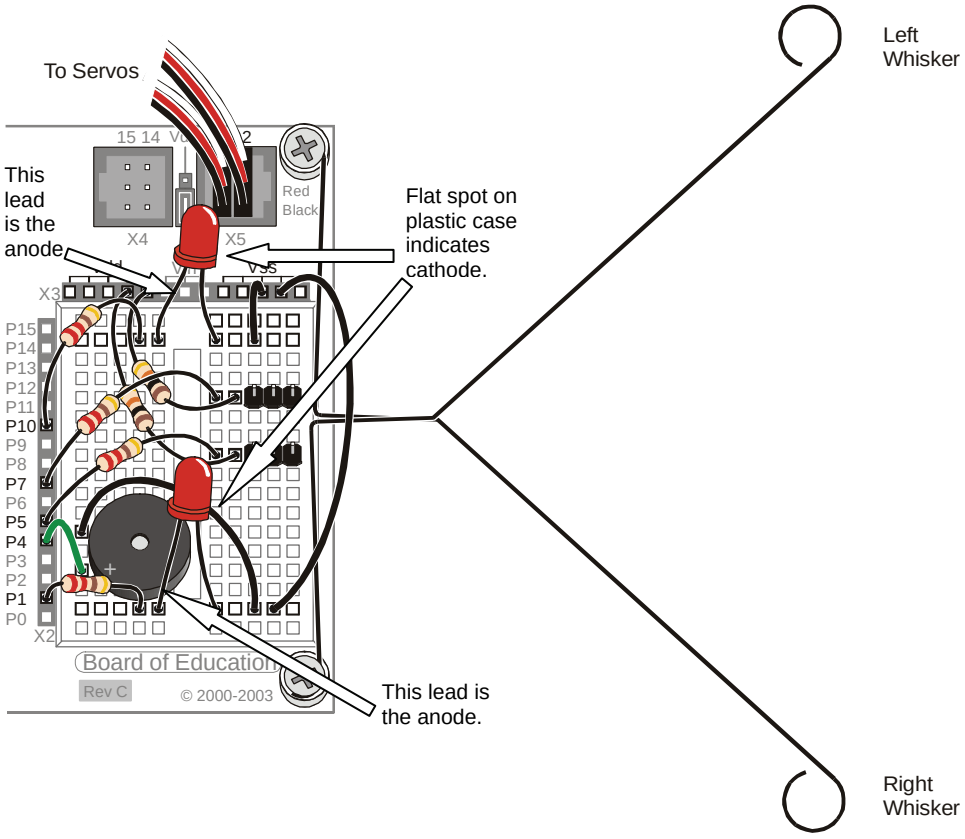
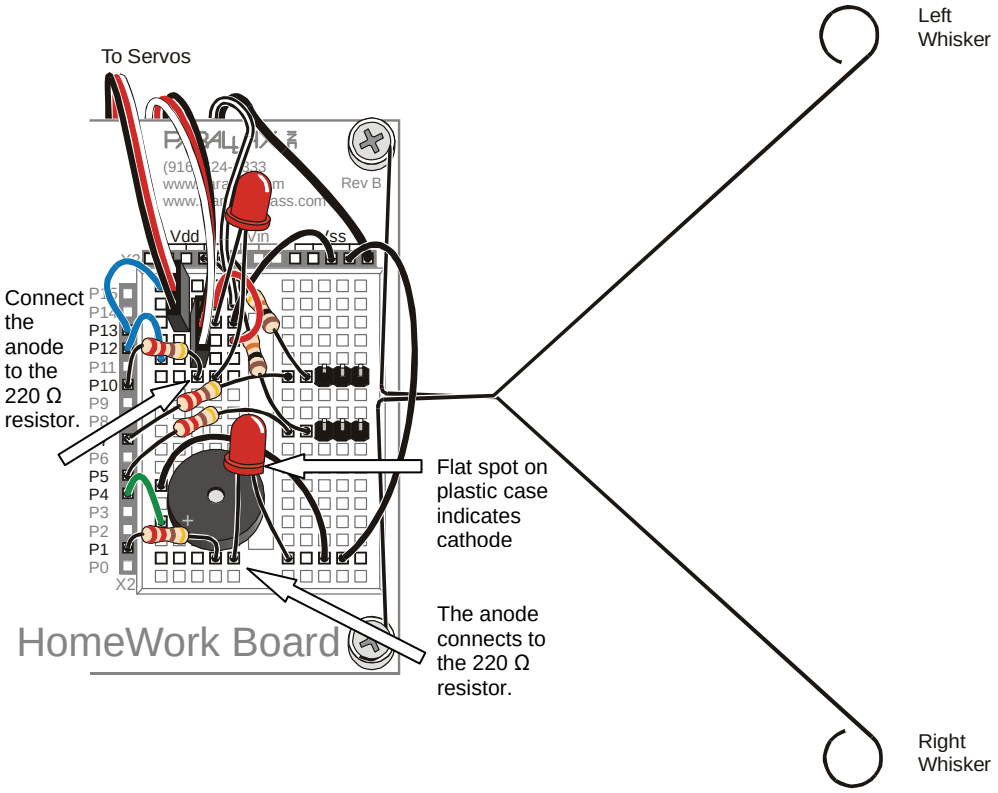


Figure 5-11 Whisker Plus LED Wiring Diagram for the HomeWork Board



Programming the LED Whisker Testing Circuits

- ✓ Reconnect power to your board.
- ✓ Save TestWhiskers.bs2 as TestWhiskersWithLeds.bs2.
- ✓ Insert these two **`IF...THEN`** statements between the **`PAUSE 50`** and **`LOOP`** commands.

```
IF (IN7 = 0) THEN
  HIGH 1
ELSE
  LOW 1
ENDIF
```

```
IF (IN5 = 0) THEN
  HIGH 10
ELSE
  LOW 10
ENDIF
```

These are called **IF…THEN** statements, and they will be more fully introduced in the next activity. These statements are used to make decisions in PBASIC. The first of the two **IF…THEN** statements sets P1 high, which turns the LED on when the whisker connected to P7 is pressed (**IN7** = 0). The **ELSE** portion of the statement makes P1 go low, which turns the LED off when the whisker is not pressed. The second **IF…THEN** statement does the same thing for the whisker connected to P5 and the LED connected to P10.

- ✓ RunTestWhiskersWithLeds.bs2.
- ✓ Test the program by gently pressing the whiskers. The red LEDs should light up when each whisker has made contact with its 3-pin header.

ACTIVITY #3: NAVIGATION WITH WHISKERS

In the previous activity, the BASIC Stamp was programmed to detect whether a given whisker was pressed. In this activity, the BASIC Stamp will be programmed to take advantage of this information to guide the Boe-Bot. When the Boe-Bot is rolling along and a whisker is pressed, it means the Boe-Bot bumped into something. A navigation program needs to take this input, decide what it means, and call a set of maneuvers that will make the Boe-Bot back up from the obstacle, turn, and go in a different direction.

Programming the Boe-Bot to Navigate Based on Whisker Inputs

This next program makes the Boe-Bot go forward until it encounters an obstacle. In this case, the Boe-Bot knows when it encounters an obstacle by bumping into it with one or both of its whiskers. As soon as the obstacle is detected by the whiskers, the navigation routines and subroutines developed in Chapter 4 will make the Boe-Bot back up and turn. Then, the Boe-Bot resumes forward motion until it bumps into another obstacle.

In order to do that, the Boe-Bot needs to be programmed to make decisions. PBASIC has a command called an **IF…THEN** statement that makes decisions. The syntax for **IF…THEN** statements is:

IF *(condition)* THEN…{ELSEIF *(condition)*}…{ELSE}…ENDIF

The "…" means you can place a code block (one or more commands) between the keywords. The next example program makes decisions based on the whisker inputs, and then calls subroutines to make the Boe-Bot take action. The subroutines are similar to the ones you developed in Chapter 4. Here is how **IF…THEN** is used.

```
IF (IN5  = 0) AND (IN7 = 0) THEN
  GOSUB Back_Up                 ' Both whiskers detect obstacle,
  GOSUB Turn_Left               ' back up & U-turn (left twice)
  GOSUB Turn_Left
ELSEIF (IN5  = 0) THEN          ' Left whisker contacts
  GOSUB Back_Up                 ' Back up & turn right
  GOSUB Turn_Right
ELSEIF (IN7  = 0) THEN          ' Right whisker contacts
  GOSUB Back_Up                 ' Back up & turn left
  GOSUB Turn_Left
ELSE                            ' Both whiskers 1, no contacts
  GOSUB Forward_Pulse           ' Apply a forward pulse &
ENDIF                           ' check again
```

Example Program: RoamingWithWhiskers.bs2

This program demonstrates one way of evaluating the whisker inputs and deciding which navigation subroutine to call using **IF…THEN**.

- ✓ Reconnect power to your board and servos.
- ✓ Enter, save, and run RoamingWithWhiskers.bs2.
- ✓ Try letting the Boe-Bot roam. When it contacts obstacles in its path, it should back up, turn, and then roam in a new direction.

```
' -----[ Title ]-------------------------------------------------------------
' Robotics with the Boe-Bot - RoamingWithWhiskers.bs2
' Boe-Bot uses whiskers to detect objects, and navigates around them.

' {$STAMP BS2}                             ' Stamp directive.
' {$PBASIC 2.5}                            ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Variables ]----------------------------------------------------------

pulseCount     VAR     Byte                ' FOR...NEXT loop counter.

' -----[ Initialization ]-----------------------------------------------------

FREQOUT 4, 2000, 3000                      ' Signal program start/reset.
```

```
' -----[ Main Routine ]-------------------------------------------------------

DO
  IF (IN5  = 0) AND (IN7 = 0) THEN           ' Both whiskers detect obstacle
    GOSUB Back_Up                            ' Back up & U-turn (left twice)
    GOSUB Turn_Left
    GOSUB Turn_Left
  ELSEIF (IN5  = 0) THEN                     ' Left whisker contacts
    GOSUB Back_Up                            ' Back up & turn right
    GOSUB Turn_Right
  ELSEIF (IN7  = 0) THEN                     ' Right whisker contacts
    GOSUB Back_Up                            ' Back up & turn left
    GOSUB Turn_Left
  ELSE                                       ' Both whiskers 1, no contacts
    GOSUB Forward_Pulse                      ' Apply a forward pulse
  ENDIF                                      ' and check again
LOOP

' -----[ Subroutines ]---------------------------------------------------------

Forward_Pulse:                               ' Send a single forward pulse.
  PULSOUT 13,850
  PULSOUT 12,650
  PAUSE 20
  RETURN

Turn_Left:                                   ' Left turn, about 90-degrees.
  FOR pulseCount = 0 TO 20
    PULSOUT 13, 650
    PULSOUT 12, 650
    PAUSE 20
  NEXT
  RETURN

Turn_Right:
  FOR pulseCount = 0 TO 20                   ' Right turn, about 90-degrees.
    PULSOUT 13, 850
    PULSOUT 12, 850

    PAUSE 20
  NEXT
  RETURN

Back_Up:                                     ' Back up.
  FOR pulseCount = 0 TO 40
    PULSOUT 13, 650
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN
```

How Roaming with Whiskers Works

The **IF...THEN** statements in the Main Routine section first check the whiskers for any states that require attention. If both whiskers are pressed (**IN5** = 0 and **IN7** = 0), a U-turn is executed by calling the **Back_Up** subroutine followed by calling the **Turn_Left** subroutine twice in a row. If just the left whisker is pressed (**IN5** = 0), then the program calls the **Back_Up** subroutine followed by the **Turn_Right** subroutine. If the right whisker is pressed (**IN7** = 0), the **Back_Up** subroutine is called, followed by the **Turn_Left** subroutine. The only possible combination that has not been covered is if neither whisker is pressed (**IN5** = 1 and **IN7** = 1). The **ELSE** command calls the **Forward_Pulse** subroutine in this case.

```
IF (IN5  = 0) AND (IN7 = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Left
  GOSUB Turn_Left
ELSEIF (IN5  = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Right
ELSEIF (IN7  = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Left
ELSE
  GOSUB Forward_Pulse
ENDIF
```

The **Turn_Left**, **Turn_Right**, and **Back_Up** subroutines should look fairly familiar, but the **Forward_Pulse** subroutine has a twist. It just sends one pulse, then returns. This is really important, because it means the Boe-Bot can check its whiskers between each forward pulse. That means the Boe-Bot checks for obstacles roughly 40 times per second as it travels forward.

```
Forward_Pulse:
  PULSOUT 12,650
  PULSOUT 13,850
  PAUSE 20
  RETURN
```

Since each full speed forward pulse makes the Boe-Bot roll around half a centimeter, it's a really good idea to only send one pulse, then go back and check the whiskers again. Since the **IF…THEN** statement is inside a **DO…LOOP**, each time the program returns from a

Forward_Pulse, it gets to **LOOP**, which sends the program back up to **DO**. What happens then? The **IF…THEN** statement checks the whiskers all over again.

Your Turn

The **FOR...NEXT** loop ***EndValue*** arguments in the **Back_Right** and **Back_Left** routines can be adjusted for more or less turn, and the **Back_Up** routine can have its ***EndValue*** adjusted to back up less for navigation in tighter spaces.

- ✓ Experiment with the **FOR...NEXT** loop ***EndValue*** arguments in the navigation routines in RoamingWithWhiskers.bs2.

You can also modify your **IF…THEN** statements to make the LED indicators from the previous activity broadcast what maneuver the Boe-Bot is in by adding **HIGH** and **LOW** commands to control the LED circuits. Here is an example.

```
IF (IN5  = 0) AND (IN7 = 0) THEN
  HIGH 10
  HIGH 1
  GOSUB Back_Up
  GOSUB Turn_Left
  GOSUB Turn_Left
ELSEIF (IN5  = 0) THEN
  HIGH 10
  GOSUB Back_Up
  GOSUB Turn_Right
ELSEIF (IN7  = 0) THEN
  HIGH 1
  GOSUB Back_Up
  GOSUB Turn_Left
ELSE
  LOW 10
  LOW 1
  GOSUB Forward_Pulse
ENDIF
```

- ✓ Modify the **IF…THEN** statement in RoamingWithWhiskers.bs2 to make the Boe-Bot broadcast its maneuver using the LED indicators.

ACTIVITY #4: ARTIFICIAL INTELLIGENCE AND DECIDING WHEN YOU'RE STUCK

You may have noticed that the Boe-Bot gets stuck in corners. As the Boe-Bot enters the corner, its whisker touches the wall on the left, so it turns right. When the Boe-Bot moves forward again, its right whisker bumps the wall on the right, so it turns left. Then it turns and bumps the left wall again, and the right wall again, and so on, until somebody rescues it from its predicament.

Programming to Escape Corners

RoamingWithWhiskers.bs2 can be modified to detect this problem and act upon it. The trick is to count the number of times that alternate whiskers are contacted. One important thing about this trick is that the program has to remember what state each whisker was in during the previous contact. It has to compare that to the whisker states of the current contact. If they are opposite, then add one to the counter. If the counter goes over a threshold that you (the programmer) have determined, then, it's time to do a U-turn and reset that alternate whisker counter.

This next program also relies on the fact that you can "nest" **IF…THEN** statements. In other words, the program checks for one condition, and if that condition is true, it checks for another condition within the first condition. Here is a pseudo code example of how it can be used.

```
IF condition1 THEN
  Commands for condition1
  IF condition2 THEN
    Commands for both condition2 and condition1
  ELSE
    Commands for condition1 but not condition2
  ENDIF
  ELSE
    Commands for not condition1
ENDIF
```

There is an example of nested **IF…THEN** statements in the routine that detects consecutive alternate whisker contacts in the next program.

Example Program: EscapingCorners.bs2

This program will cause your Boe-Bot to execute a U-turn at either the fourth or fifth alternate corner, depending on which whisker was pressed first.

- ✓ Enter, save, and run EscapingCorners.bs2.
- ✓ Test this program by pressing alternate whiskers as the Boe-Bot roams. Depending on which whisker you started with, the Boe-Bot should execute its U-Turn maneuver after either the fourth or fifth consecutive whisker press.


```
' -----[ Title ]-----------------------------------------------------------
' Robotics with the Boe-Bot - EscapingCorners.bs2
' Boe-Bot navigates out of corners by detecting alternating whisker presses.
' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Variables ]-------------------------------------------------------

pulseCount     VAR     Byte                ' FOR...NEXT loop counter.
counter        VAR     Nib                 ' Counts alternate contacts.
old7           VAR     Bit                 ' Stores previous IN7.
old5           VAR     Bit                 ' Stores previous IN5.

' -----[ Initialization ]--------------------------------------------------

FREQOUT 4, 2000, 3000                      ' Signal program start/reset.
counter = 1                                ' Start alternate corner count.
old7 = 0                                   ' Make up old values.
old5 = 1

' -----[ Main Routine ]----------------------------------------------------

DO

' ---  Detect Consecutive Alternate Corners -----------------------
' See the "How EscapingCorners.bs2 Works" section that follows this program.

  IF (IN7 <> IN5) THEN                     ' One or other is pressed.
    IF (old7 <> IN7) AND (old5 <> IN5) THEN  ' Different from previous.
      counter = counter + 1                ' Alternate whisker count + 1.
      old7 = IN7                           ' Record this whisker press
      old5 = IN5                           ' for next comparison.
      IF (counter > 4) THEN                ' If alternate whisker count = 4,
        counter = 1                        ' reset whisker counter
        GOSUB Back_Up                      ' and execute a U-turn.
        GOSUB Turn_Left
        GOSUB Turn_Left
      ENDIF                                ' ENDIF counter > 4.
    ELSE                                   ' ELSE (old7=IN7) or (old5=IN5),
      counter = 1                          ' not alternate, reset counter.
    ENDIF                                  ' ENDIF (old7<>IN7) and
                                           ' (old5<>IN5).
  ENDIF                                    ' ENDIF (IN7<>IN5).
```

```
' ---  Same navigation routine from RoamingWithWhiskers.bs2 ------------------

  IF (IN5  = 0) AND (IN7 = 0) THEN           ' Both whiskers detect obstacle
    GOSUB Back_Up                            ' Back up & U-turn (left twice)
    GOSUB Turn_Left
    GOSUB Turn_Left
  ELSEIF (IN5  = 0) THEN                     ' Left whisker contacts
    GOSUB Back_Up                            ' Back up & turn right
    GOSUB Turn_Right
  ELSEIF (IN7  = 0) THEN                     ' Right whisker contacts
    GOSUB Back_Up                            ' Back up & turn left
    GOSUB Turn_Left
  ELSE                                       ' Both whiskers 1, no contacts
    GOSUB Forward_Pulse                      ' Apply a forward pulse
  ENDIF                                      ' and check again

LOOP

' -----[ Subroutines ]----------------------------------------------------

Forward_Pulse:                               ' Send a single forward pulse.
  PULSOUT 13,850
  PULSOUT 12,650
  PAUSE 20
  RETURN

Turn_Left:                                   ' Left turn, about 90-degrees.
  FOR pulseCount = 0 TO 20
    PULSOUT 13, 650
    PULSOUT 12, 650
    PAUSE 20
  NEXT
  RETURN

Turn_Right:
  FOR pulseCount = 0 TO 20                   ' Right turn, about 90-degrees.
    PULSOUT 13, 850
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN

Back_Up:                                     ' Back up.
  FOR pulseCount = 0 TO 40
    PULSOUT 13, 650
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN
```

How EscapingCorners.bs2 Works

Since this program is a modified version of RoamingWithWhiskers.bs2, only new features related to detecting and escaping corners are discussed here.

Three extra variables are created for detecting a corner. The nibble variable **counter** can store a value between 0 and 15. Since our target value for detecting a corner is 4, the size of the variable is reasonable. Remember that a bit variable can store a single bit, either a 1 or a 0. The next two variables (**old7** and **old5**) are both bit variables. These are also the right size for the job since they are used to store old values of **IN7** and **IN5**, which are also bit variables.


```
counter         VAR     Nib
old7            VAR     Bit
old5            VAR     Bit
```

These variables have to be initialized (given initial values). For the sake of making the program easier to read, **counter** is set to 1, and when it gets to 4 due to the fact that the Boe-Bot is stuck in a corner, it is reset to 1. The **old7** and **old5** variables have to be set so that it looks like one of the two whiskers was pressed some time before the program started. This has to be done because the routine for detecting alternate corners always compares an alternating pattern, either (**IN5** = 1 and **IN7** = 0) or (**IN5** = 0 and **IN7** = 1). Likewise, **old5** and **old7** have to be different from each other.

```
counter = 1
old7 = 0
old5 = 1
```

Now we get to the Detect Consecutive Alternate Corners section. The first thing we want to check for is if one or the other whisker is pressed. A simple way to do this is to ask "is **IN7** different from **IN5**?" In PBASIC, we can use the not-equal operator <> in an **IF** statement:

```
IF (IN7 <> IN5) THEN
```

If it is indeed one whisker that is pressed, the next thing to check for is whether or not it's the exact opposite pattern as the previous time. In other words, is (**old7 <> IN7**) and is (**old5 <> IN5**)? If that's true, then, it's time to add one to the counter that tracks alternate whisker contacts. It's also time to remember the current whisker pattern by setting **old7** equal to the current **IN7** and **old5** equal to the current **IN5**.

```
          IF (old7 <> IN7) AND (old5 <> IN5) THEN
            counter = counter + 1
            old7 = IN7
            old5 = IN5
```

If it turns out that this is the fourth consecutive whisker contact, then it's time to reset **counter** to 1 and execute a U-turn.

```
            IF (counter > 4) THEN
              counter = 1
              GOSUB Back_Up
              GOSUB Turn_Left
              GOSUB Turn_Left
```

This **ENDIF** ends the code block that is executed if **counter** > 4.

```
            ENDIF
```

This **ELSE** statement is connected to the **IF (old7 <> IN7) AND (old5 <> IN5) THEN** statement. The **ELSE** statement covers what happens if the **IF** statement is not true. In other words, it must not be an alternate whisker that was pressed, so reset **counter** because the Boe-Bot is not stuck in a corner.

```
          ELSE
            counter = 1
```

This **ENDIF** statement ends the decision making process for the **IF (old7 <> IN7) AND (old5 <> IN5) THEN** statement.

```
          ENDIF
        ENDIF
```

The remainder of the program is the same as before.

Your Turn

One of the **IF...THEN** statements in EscapingCorners.bs2 checks to see if **counter** has reached 4.

- ✓ Try increasing the value to 5 and 6 and note the effect.
- ✓ Try also reducing the value and see if it has any effect on normal roaming.

SUMMARY

In this chapter, instead of navigating from a pre-programmed list, the Boe-Bot was programmed to navigate based on sensory inputs. The sensory inputs used in this chapter were whiskers, which served as normally open contact switches. When properly wired, these switches can show one voltage (5 V) at the switch's contact point when it's open and a different voltage (0 V) when it's closed. The BASIC Stamp I/O pin's input registers store "1" if they detect Vdd (5 V) and "0" if they detect Vss (0 V).

The BASIC Stamp was programmed to test the whisker sensors and display the test results using two different media, the Debug Terminal and LEDs. PBASIC programs were developed to make the BASIC Stamp check the whiskers between each servo pulse. Based on the state of the whiskers, **`IF…THEN`** statements in the program's Main Routine section called navigation subroutines similar to the ones developed in the previous chapter to guide the Boe-Bot away from obstacles. As a simple example of artificial intelligence, an additional routine was developed that enabled the Boe-Bot to detect when it got stuck in a corner. This routine involved storing old whisker states, comparing them against the current whisker states, and counting the number of alternate object detections.

This chapter introduced sensor-based Boe-Bot navigation. The next three chapters will focus on using different types of sensors to give the Boe-Bot vision. Both vision and touch open up lots of opportunities for the Boe-Bot to navigate in increasingly complex environments.

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 occur in the input register? If I/O pin P8 is used to monitor the input pin, what value does **`IN8`** have when a whisker is pressed, and what value does it have when a whisker is not pressed?
3. If **`IN7`** = 1, what does that mean? What does it mean if **`IN7`** = 0? How about **`IN5`** = 1 and **`IN5`** = 0?
4. What command is used to jump to different subroutines depending on the value of a variable? What command is used to decide which subroutine to jump to? What are these decisions based on?
5. What is the purpose of having nested **`IF…THEN`** statements?

Exercises

1. Write a **DEBUG** command for TestWhiskers.bs2 that updates each whisker state on a new line. Adjust the **PAUSE** command so that it is 250 instead of 50.
2. Using RoamingWithWhiskers.bs2 as a reference, write a **Turn_Away** subroutine that calls the **Back_Up** subroutine once and the **Turn_Left** subroutine twice. Write down the modifications you will have to make to the Main Routine section of RoamingWithWhiskers.bs2

Projects

1. Modify RoamingWithWhiskers.bs2 so that the Boe-Bot makes a 4 kHz beep that lasts 100 ms before executing the evasive maneuver. Make it beep twice if both whisker contacts are detected during the same sample.
2. Modify RoamingWithWhiskers.bs2 so that the Boe-Bot roams in a 1 yard (or meter) diameter circle. When you touch one whisker, it will cause the Boe-Bot to travel in a tighter circle (smaller diameter). When you touch the other whisker, it will cause the Boe-Bot to navigate in a wider diameter circle.

Solutions

Q1. A tactile switch.

Q2. Zero (0) volts, resulting in binary zero (0) at the input register.
IN8 = 0 when whisker is pressed.
IN8 = 1 when whisker is not pressed.

Q3. **IN7** = 1 means the right whisker is not pressed.
IN7 = 0 means the right whisker is pressed.
IN5 = 1 means the left whisker is not pressed.
IN5 = 0 means the left whisker is pressed.

Q4. The **GOSUB** command performs the actual jump. The **IF...THEN** command is used to decide which subroutine to jump to. That decision is based on conditions, which are logical statements that evaluate to true or false.

Q5. The program can check for one condition, and if that condition is true, it can check for another condition within the first condition.

E1. The key to solving this problem is to use a second **CRSRXY** command that will place the right whisker state in the proper place on the screen. To line up with the headings, the text should start on column 9 of row 3.

```
' Robotics with the Boe-Bot - TestWhiskers_UpdateEaOnNewLine.bs2
' Update each whisker state on a new line.
' {$STAMP BS2}                                ' Stamp directive.
' {$PBASIC 2.5}                               ' PBASIC directive.

DEBUG "WHISKER STATES", CR,
      "Left     Right", CR,
      "------   ------"
DO
  DEBUG CRSRXY, 0, 3, "P5 = ", BIN1 IN5       ' Print in Column 0,Row 3
  DEBUG CRSRXY, 9, 3, "P7 = ", BIN1 IN7       ' Print in Column 9,Row 3
  PAUSE 250                                   ' Change from 50 to 250
LOOP
```


E2. Subroutine:

```
Turn_Away:
  GOSUB Back_Up
  GOSUB Turn_Left
  GOSUB Turn_Left
  RETURN
```

To modify the Main Routine, replace the three **GOSUB** commands under the first **IF** condition with this single line:

```
    GOSUB Turn_Away
```

P1. The key to solving this problem is to write a statement that makes a beep with the required parameters:

```
  FREQOUT 4, 100, 4000        ' 4kHz beep for 100ms
```

This statement must be added to the Main Routine in the appropriate places, as shown below. The rest of the program is unchanged.

```
' -----[ Main Routine ]-----------------------------------------

DO
  IF (IN5  = 0) AND (IN7 = 0) THEN  ' Both whiskers detect
    FREQOUT 4, 100, 4000            ' 4 kHz beep for 100 ms
    FREQOUT 4, 100, 4000            '   Repeat twice
    GOSUB Back_Up                   ' Back up & U-turn
GOSUB Turn_Left
    GOSUB Turn_Left
  ELSEIF (IN5  = 0) THEN            ' Left whisker contacts
    FREQOUT 4, 100, 4000            ' 4 kHz beep for 100 ms
    GOSUB Back_Up                   ' Back up & turn right
    GOSUB Turn_Right
```

```
  ELSEIF (IN7  = 0) THEN            ' Right whisker contacts
    FREQOUT 4, 100, 4000            ' 4 kHz beep for 100 ms
    GOSUB Back_Up                   ' Back up & turn left
    GOSUB Turn_Left
  ELSE                              ' Both whiskers 1, no
    GOSUB Forward_Pulse             ' contacts
  ENDIF                             ' Apply a forward pulse
LOOP                                ' and check again
```

P2. We found from Chapter 4 Projects that a 1 yard circle can be achieved with **`PULSOUT 13, 850`** and **`PULSOUT 12, 716`**. Using these values as the 1 yard circle, the radius can be adjusted by slightly increasing or decreasing the pulse width from the starting value of 716. Each time a whisker is pressed the program will add or subtract a bit from the right wheel's pulse width.

```
' Robotics with the Boe-Bot - CirclingWithWhiskerInput.bs2
' Move in 1 yard circle, increase/decrease radius in response
' to whisker presses, one whisker increases, one decreases.
' {$STAMP BS2}                       ' Stamp directive.
' {$PBASIC 2.5}                      ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Variables/Initialization ]----------------------------------
pulseWidth     VAR     Word           ' Signal sent to servo
toneFreq       VAR     Word           ' Frequency of beeping tone
pulseWidth = 716                      ' Found in Ch4 to make 1y circle
toneFreq = 4000                       ' Beginning tone is 4 kHz

' -----[ Main Routine ]-----------------------------------------------

DO
  PULSOUT 13, 850                     ' Pulse servos in circular path
  PULSOUT 12, pulseWidth              ' 12 slower than 13 so it arcs
  PAUSE 20
  IF (IN5  = 0) THEN                  ' Left whisker makes circle
    IF (pulseWidth <= 845) THEN       ' smaller, down to servo max
      pulseWidth = pulseWidth + 5     ' pulseWidth of 850.
      toneFreq = toneFreq + 100
      FREQOUT 4, 100, toneFreq        ' Play tone as indicator.
    ENDIF
  ELSEIF (IN7  = 0) THEN              ' Right whisker makes circle
    IF (pulseWidth >= 655) THEN       ' larger, down to servo min
      pulseWidth = pulseWidth - 5     ' pulseWidth of 650.
      toneFreq = toneFreq - 100
      FREQOUT 4, 100, toneFreq        ' Play tone as indicator.
    ENDIF
  ENDIF
LOOP
```

Chapter 6: Light-Sensitive Navigation with Phototransistors

Should I save this chapter for last? Many classes skip to Chapter 7 and 8, and then return here if time permits. Chapter 7 is the best "next step" after navigation with whiskers because it introduces a sensor that the Boe-Bot can use to detect obstacles without bumping into them. Chapter 8 uses that same sensor for distance detection and following objects. That will complete your introduction to object detection and navigation. After that, return here to make your Boe-Bot detect and respond to something entirely different and somewhat more challenging—ambient light.

Download select example code: Some of the longer example programs in this chapter are available for download from www.parallax.com/go/Boe-Bot. Look for the file LightSensorExamples.zip.

Light has many applications in robotics and industrial control. Some examples include sensing the edge of a roll of fabric in the textile industry, 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.

There are many different light sensors that serve unique functions. The light sensors in your Boe-Bot kit respond to visible light along with an invisible type of light called infrared. These sensors can be incorporated into a couple of different circuits, and the BASIC Stamp can be programmed to interact with them to detect variations in light level. With this information, your program can be expanded to make the Boe-Bot recognize areas with light or dark perimeters, report overall brightness and darkness levels, and seek out light sources such as flashlight beams and doorways that are letting light into dark rooms.

INTRODUCING THE PHOTOTRANSISTOR

A transistor is like a valve that regulates the amount of electric current that passes through two of its terminals. The third terminal of a transistor 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.

Figure 6-1 shows the schematic and part drawing of the phototransistor in your Boe-Bot Robot 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.

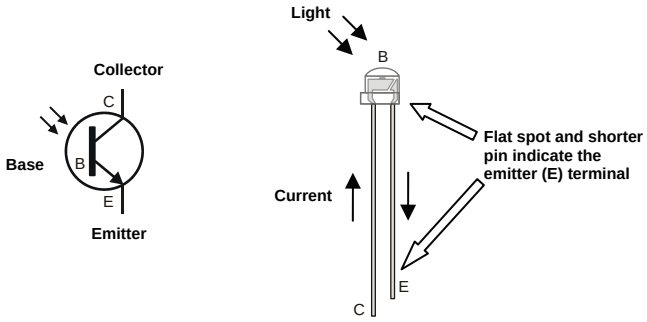


Figure 6-1
Phototransistor Schematic Symbol and Part Drawing

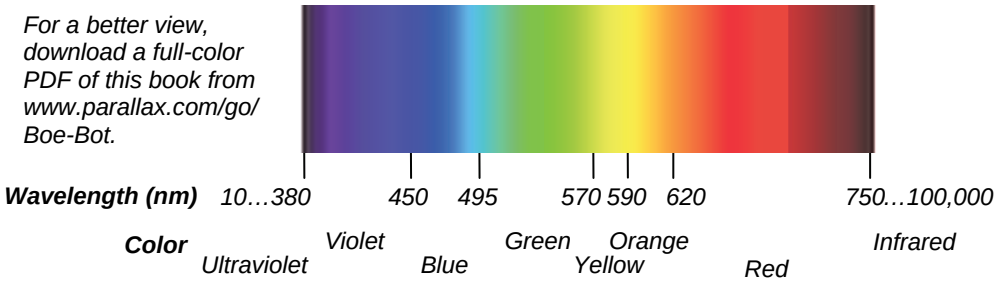
Although the phototransistor and LED are different devices, they do have two similarities. First, if you connect the phototransistor in the circuit backwards, it won't work right. Second, the phototransistor has two different length pins and a flat spot on its plastic case for identifying its terminals. The longer of the two pins indicates the phototransistor's collector terminal, and the shorter pin indicates the emitter. The emitter terminal also connects closer to a flat spot on the phototransistor's clear plastic case, which is useful for identifying the terminals if the leads have been trimmed.

- ✓ Check Figure 6-1 and find the emitter terminal's flat spot and shorter pin.

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. Figure 6-2 shows the wavelengths for colors of light we are familiar with along with some the human eye cannot detect, such as ultraviolet and infrared.

The phototransistor in the Boe-Bot Parts Kit has its peak sensitivity at 850 nm, which according to Figure 6-2, 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. The phototransistor also responds to visible light, thought it's somewhat less sensitive, especially to wavelengths below 450 nm, which are left of blue in the figure.

Figure 6-2 Wavelengths and their Corresponding Colors



Circuit designs that use phototransistors for light detection can be adjusted to perform better in certain light levels, and the phototransistor circuits in this chapter are designed for indoor use. So if your robotics area has fluorescent, incandescent, or indirect halogen indoor lighting, it should work great. Avoid sunlight streaming in through nearby windows, because it'll flood the phototransistors with too much infrared light. If your work area is near windows that let sunlight in, it's a good idea to draw the blinds before getting started. Halogen lamps pointed directly at the course could also cause problems. They should only provide indirect light, ideally directed upward so that the light is reflected off the ceiling. For best results, set up your course in an area with bright fluorescent lighting.

Illuminance is a scientific name for the measurement of incident light. One way to understand incident light is to think about shining a flashlight at a wall. The focused beam that you see is incident light. The unit of measurement of luminance is commonly the "foot-candle" in the English system or the "lux" in the metric system. Boe-Bot phototransistor measurements won't be concerned with lux levels, just whether incident light coming from certain directions is brighter or darker. The Boe-Bot's program can then use differences between right and left illuminance levels to make navigation decisions.

ACTIVITY #1: A SIMPLE BINARY LIGHT SENSOR

Imagine that your Boe-Bot is navigating a course, and that there's a bright light at the end. For example, it could be a bright light pointing down at a certain spot. Your Boe-Bot's last task in the course could be to stop underneath that bright light. 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.

If stopping under a bright light is your Boe-Bot's only light-seeking task, there's a simple circuit you can use that lets the BASIC Stamp know it detected bright light with a binary-1, or ambient light with a binary-0.

Ambient According to Merriam Webster's Dictionary, the word *ambient* means existing or present on all sides. 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, 220 Ω (red-red-brown)
(1) Resistor, 470 Ω (yellow-violet-brown)
(1) Resistor, 1 kΩ (brown-black-red)
(1) Resistor, 2 kΩ (red-black-red)
(1) Resistor, 4.7 kΩ (yellow-violet-red)
(1) Resistor, 10 kΩ (brown-black-orange)

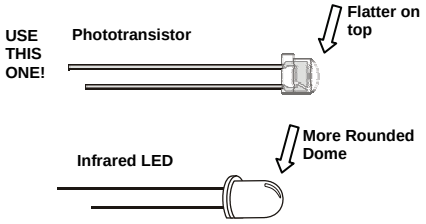


Figure 6-3: Phototransistors vs. IR LEDs

Building the Bright Light Detector

Figure 6-4 shows the schematic and wiring diagram of a voltage output phototransistor circuit that the BASIC Stamp will use to get that binary 1 or 0 value. 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 the other resistors in the parts list.

The circuit in Figure 6-4 is similar to ones you will find in lights that automatically turn on at night, and certain conveyer belt detectors.

- ✓ Disconnect power from your board and servos.
- ✓ Build the circuit shown in Figure 6-4.
- ✓ The wiring diagram points out the phototransistor emitter pin, which is shorter and closer to the flat spot on the plastic case. Double check your wiring using the figure as a reference to make sure the phototransistor's collector and emitter are correctly connected in your light sensing circuits.

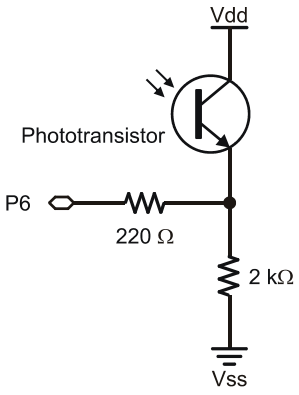
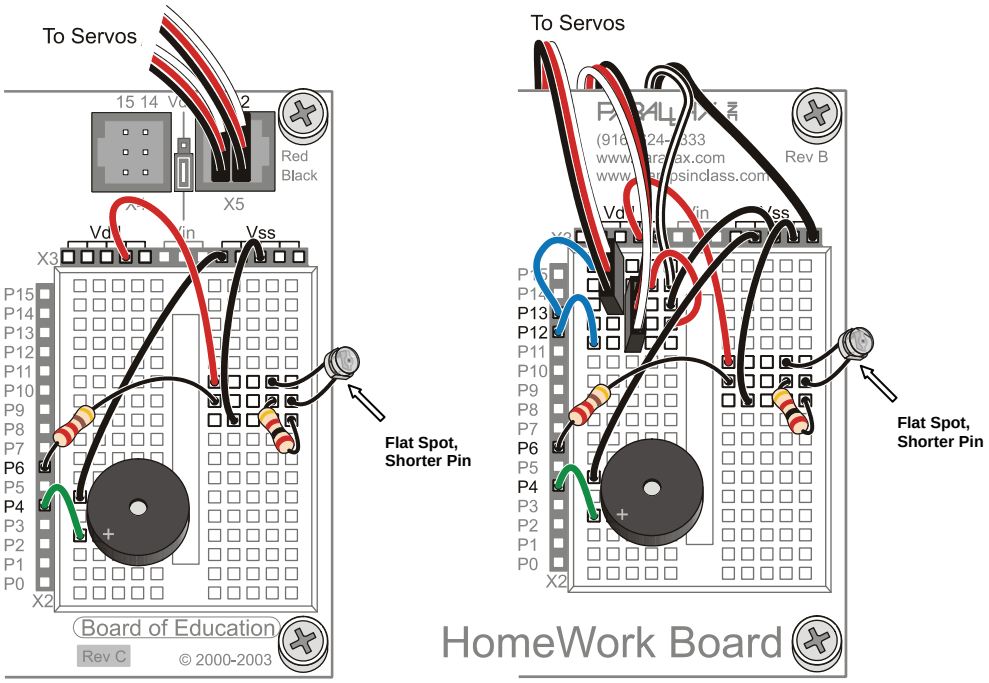


Figure 6-4
Phototransistor Voltage Output Circuit

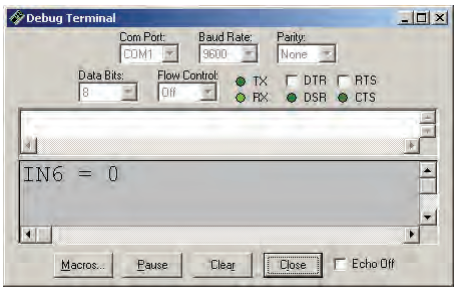
Wiring Diagrams
Board of Education (left);
HomeWork Board (right)



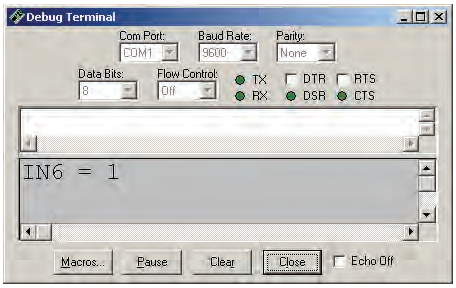
Example Program: TestBinaryPhototransistor.bs2

This program should make the Debug Terminal display a value of 0 in a room with fluorescent lights, and no direct sunlight. When you shine a bright light on the phototransistor, the program should display a value of 1. Figure 6-5 shows an example.

Figure 6-5: Debug Terminal Displays TestBinaryPhototransistor.bs2 Messages



Ambient Fluorescent Light



Bright Light

- ✓ Make sure the phototransistor leads do not touch each other. Optionally wrap the exposed portions of the leads with electrical tape.
- ✓ Reconnect power to your board.
- ✓ Enter, save, and run TestBinaryPhototransistor.bs2.
- ✓ Watch the value of **`IN6`** in the Debug Terminal, and verify that it stores a 0 when it's not under the bright light and a 1 when it's under bright light. Good sources of bright light include incandescent flashlights (flashlights with bulbs, not LEDs), incandescent desk lamps, and small halogen lamps.
- ✓ If the ambient light is brighter than just fluorescent lights, and you have a nice bright lamp, 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.

```
' Robotics with the Boe-Bot - TestBinaryPhototransistor.bs2
' Display 1 when the phototransistor circuit applies more than 1.4 V to P6
' or 0 when it applies less than 1.4 V.

' {$STAMP BS2}
' {$PBASIC 2.5}

PAUSE 1000
DEBUG CLS

DO

  DEBUG HOME, "IN6 = ", BIN IN6
  PAUSE 100

LOOP
```


Your Turn – Make the Boe-Bot Halt Under the Bright Light

HaltUnderBrightLight.bs2 will make the Boe-Bot go forward until the phototransistor detects light that's bright enough to make **IN6** store a binary-1.

- ✓ Try starting the program with the Boe-Bot a few feet from the bright light.
- ✓ Point the Boe-Bot so that it will go straight under the bright light. How close did the Boe-Bot get to stopping directly under the light?
- ✓ Try making adjustments to the code and resistor values to get the Boe-Bot to park right underneath that bright light.

```
' Robotics with the Boe-Bot - HaltUnderBrightLight.bs2
' Full speed forward until bright light makes phototransistor circuit's
' output voltage exceed 1.4 V, resulting in IN6=1

' {$STAMP BS2}
' {$PBASIC 2.5}

FREQOUT 4, 2000, 3000
DEBUG "Program running... "

DO UNTIL IN6 = 1

  PULSOUT 13, 850
  PULSOUT 12, 650
  PAUSE 20

LOOP
```

Advanced Topic: How the Phototransistor Voltage Output Circuit Works

The phototransistor circuit you built applies a voltage to I/O pin P6. The voltage labeled V_{P6} in Figure 6-6 is the voltage output that the circuit applies to the I/O pin. This voltage increases with more light and decreases with less light. Since P6 is set to input, this voltage causes **`IN6`** to store a binary-1 or a binary-0. If the voltage is above 1.4 V, **`IN6`** stores a binary-1; if it's below 1.4 V, **`IN6`** stores a binary-0.

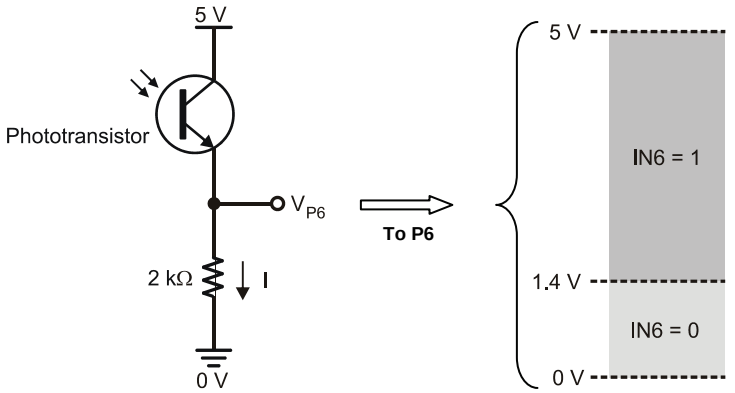


Figure 6-6
Phototransistor Voltage Output Circuit and IN6 Response to V_{P6}

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.

When a BASIC Stamp I/O pin is an input, the circuit behaves as though neither the I/O pin nor the 220 Ω resistor is present. Figure 6-6 shows a circuit that's equivalent to the one you just built on the breadboard when the I/O pin is set to input. With Vdd (5 V) at the top and ground (0 V) at the bottom of the circuit, 5 V of electrical pressure (voltage) makes the supply of electrons in the Boe-Bot's batteries want to flow through it.

Connected in Series When two or more elements are connected end-to-end, they are connected in series. The phototransistor and resistor in Figure 6-6 are connected in series.

A logic threshold is a voltage that distinguishes a binary-1 from a binary-0. For a BASIC Stamp I/O pin set to input, that threshold is 1.4 V.

The reason the voltage at P6 changes with light is because the phototransistor lets more current pass with more light, or less current pass with less light. That current, which is labeled I in Figure 6-6, also has to pass though 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 Vss = 0 V, the voltage at the V_{P6} end goes up with more current and down with less current.

If you replace the 2 kΩ resistor with a 1 kΩ resistor, V_{P6} will be smaller values for the same currents. In fact, it will take twice as much current to get V_{P6} past the BASIC Stamp I/O pin's 1.4 V logic threshold, which means the light will have to be twice as bright to make **`IN6`** to store a 1. 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_{P6} 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_{P6} increase to above 1.4 V to make **`IN6`** store a 1. So, a larger resistor makes the circuit more sensitive to light.

Ohm's Law for Calculating Voltage, Current, and Resistance

Two properties affect the voltage at V_{P6}: 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:

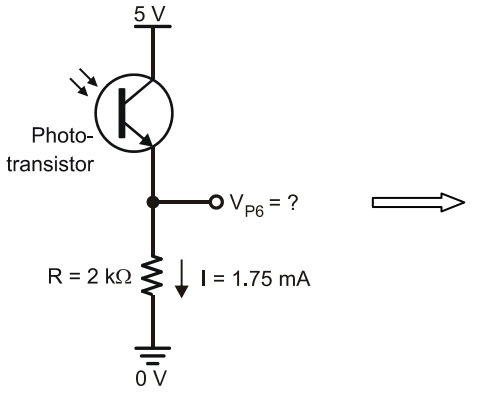
$$\mathbf{V = I \times R}$$

In some textbooks, you will see E = I × R instead. E stands for electric potential.

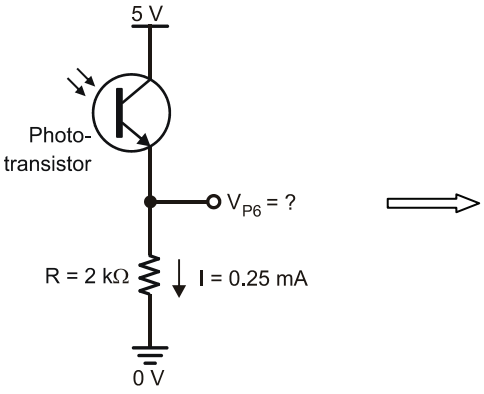
Voltage V is measured in units of volts, which are abbreviated with an upper-case V. Current I is measured in amps, which are abbreviated A, and resistance R is measured in ohms which is abbreviated with the Greek letter omega (Ω). The current levels you are likely to see through this circuit are in milliamps (mA). The lower-case m indicates that it's a measurement of thousandths of amps. Likewise, the lower-case k in kΩ indicates that the measurement is in thousands of ohms.

Let's use Ohm's Law to calculate V_{P6} 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, and 0.25 mA, which would happen with less bright light. Figure 6-8 shows the conditions and their solutions. When you try these calculations, make sure to remember that milli (m) is thousandths and kilo (k) is thousands when you substitute the numbers into Ohm's Law.

Figure 6-7: V_{P6} Calculations for Two Different Phototransistor-Resistor Currents



$$
\begin{aligned}
V_{P6} &= I \times R \\
&= 1.75\,mA \times 2\,k\Omega \\
&= \frac{1.75}{1000}\,A \times 2000\ \Omega \\
&= 1.75\,A \times 2\,\Omega \\
&= 3.5\,A\Omega \\
&= 3.5V
\end{aligned}
$$



$$
\begin{aligned}
V_{P6} &= I \times R \\
&= 0.25\,mA \times 2\,k\Omega \\
&= \frac{0.25}{1000}\,A \times 2000\ \Omega \\
&= 0.25\,A \times 2\,\Omega \\
&= 0.5\,A\Omega \\
&= 0.5V
\end{aligned}
$$

Your Turn – Ohm's Law and Resistor Adjustments

Let's say that the light in your room is twice as bright as the one in the room that resulted in Vo = 3.5 V for bright light and 0.5 V for shade. Another situation that could cause higher current is if the 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, and bright current I = 3.5 mA and dim current I = 0.5 mA. Does it bring Vo 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 WITH PHOTOTRANSISTORS

This activity introduces a circuit that the BASIC Stamp can use to measure the brightness of light incident on the phototransistor's base. The values of the measurements could range from small numbers, indicating bright light, to large numbers, indicating low 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 that 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.

Parts List

In this activity, you will need two phototransistors and two 0.1 µF capacitors. Figure 6-8 shows drawings of both.

- ✓ Look carefully at Figure 6-8 and make note of the difference between a phototransistor and an infrared LED.

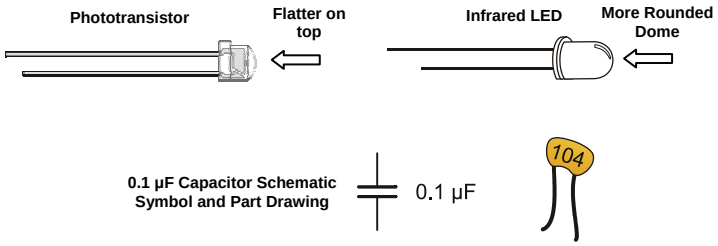


Figure 6-8
Distinguishing Phototransistors from Infrared LEDs; Identifying the 0.1 µF Capacitor

- ✓ Gather the parts listed below using Figure 6-8 as a guide for finding the phototransistors and 0.1 µF capacitors in your parts kit.

(2) Phototransistors
(2) Capacitors, 0.1 µF (104)
(2) Resistors, 1 kΩ (brown-black-red)
(2) Jumper wires

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 the capacitor tends to store is measured in farads (F). A farad is a very large value that's not practical for use with these Boe-Bot circuits. The capacitors you will use in this activity store fractions of millionths of farads. A millionth of a farad is called a microfarad, and it is abbreviated µF. The capacitor you will use in this exercise stores one tenth of a millionth of a farad. That's 0.1 µF.

Common capacitance measurements are:

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 picofarads or (pF). In this labeling system, 104 is the number 10 with four zeros added, so the capacitor is 100,000 pF, which is 0.1 µF.

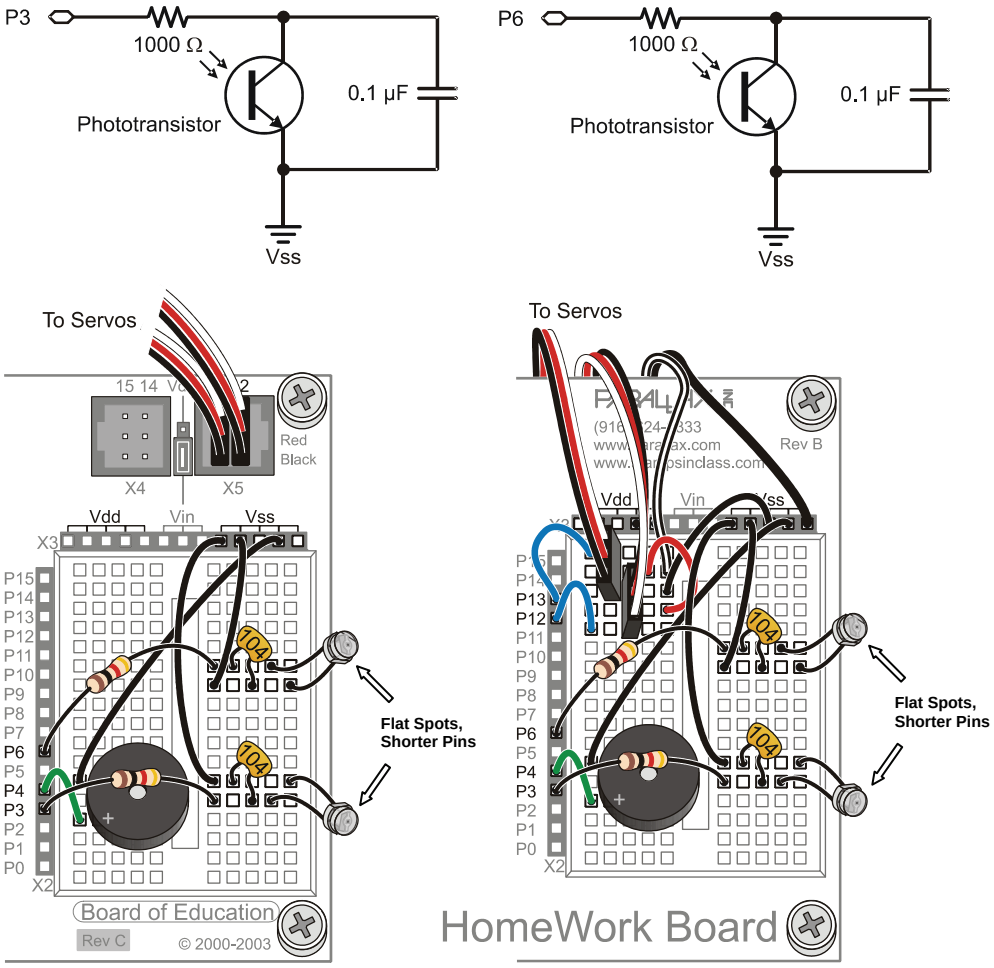
$(100{,}000) \times (1 \times 10^{-12})$ F = $(100 \times 10^{3}) \times (1 \times 10^{-12})$ F
= 100×10^{-9} F = 0.1×10^{-6} F
= 0.1 µF.

Building the Photosensitive Eyes

The BASIC Stamp can use the circuits in Figure 6-9 to measure the amount of light incident on each phototransistor's base. One phototransistor will be pointing forward and to the left, and the other will be pointing forward and to the right. They will both also be pointing upward at about 45°. Since they are pointing in different directions, the BASIC Stamp will be able to use them to determine whether light is brighter on the Boe-Bot's left or right.

- ✓ Disconnect power from your board and servos.
- ✓ Remove all the phototransistor voltage output circuit parts from Figure 6-4, including the wire that connected the phototransistor's collector terminal to Vdd.
- ✓ Build the circuits shown in Figure 6-9.
- ✓ Double check your circuits against the wiring diagram to make sure your phototransistors are not plugged in backwards. Use the "shorter pin" and "flat spot" indicators as a guide.

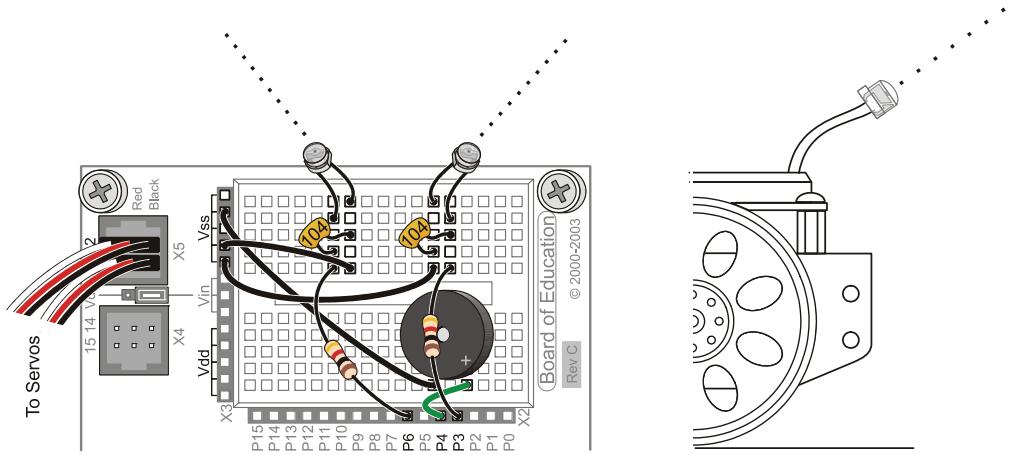
Figure 6-9: Analog Phototransistor Circuit Schematics and Wiring Diagram



The roaming examples in this chapter will depend on the phototransistors being pointed upward and outward to detect differences in incident light levels from different directions.

- ✓ Make sure your phototransistors are pointing upward and outward as shown in Figure 6-10.

Figure 6-10: Point the Phototransistors Upward and Outward



About Charge Transfer and the Phototransistor Circuit

Each phototransistor/capacitor circuit is called a charge transfer circuit. The BASIC Stamp will measure the rate at which each capacitor loses its charge through its phototransistor by measuring how long it takes the capacitor's voltage to decay. The decay time corresponds to the brightness of the light incident on the phototransistor's base. Faster decay means more light, slower decay means less light.

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.

Think of the capacitors in the Figure 6-11 circuit as tiny rechargeable batteries, and think of the phototransistors as light controlled current valves. 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 shinning on the phototransistor's base. Again, brighter light results in more current, shadows result in less current.

Connected in Parallel

The phototransistor and capacitor shown in Figure 6-11 are connected in parallel. For two components to be connected in parallel, each of their leads must be connected to common terminals (also called nodes). The phototransistor and the capacitor each have one lead connected to Vss. They also each have one lead connected to the same 1 kΩ resistor lead. So, they are connected in parallel.

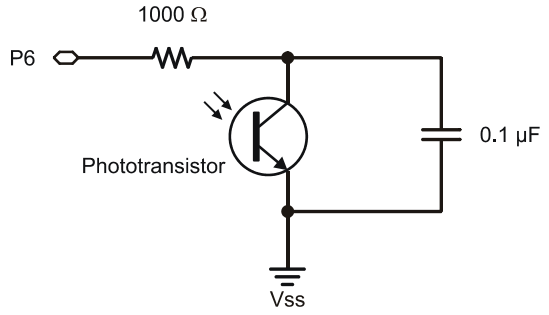


Figure 6-11
QT Circuit Connected to I/O Pin P6

The BASIC Stamp performs these steps to measure a light level with the phototransistor charge transfer circuit in Figure 6-11:

1. Use the **HIGH** command to apply 5 V to the circuit and charge the capacitor (tiny battery).
2. Use the **PAUSE** command to wait for the capacitor to charge.
3. Use the **RCTIME** command to set the I/O pin to input and measure the time it takes for the capacitor's voltage to decay to 1.4 V as it loses charge through the phototransistor.

A longer decay time measurement in step 3 means less light; a shorter decay time means more light.

The **RCTIME** command changes the ***Pin*** direction from output to input, and then waits for the I/O pin's state to change, which happens when the voltage the circuit applies to the pin passes its 1.4 V logic threshold. The **RCTIME** command stores the time measurement result in ***Variable***. With the BASIC Stamp 2, this result is a number of 2 µs increments.

RCTIME *Pin*, *State*, *Variable*

If the ***State*** argument is set to 1, **`RCTIME`** will wait for it to change to 0 indicating that the voltage decayed down to 1.4 V. If **`State`** is set to 0, **`RCTIME`** will wait for the voltage to rise to 1.4 V. In either case, the command stores the time measurement result in the ***Variable*** argument, which is typically a word variable.

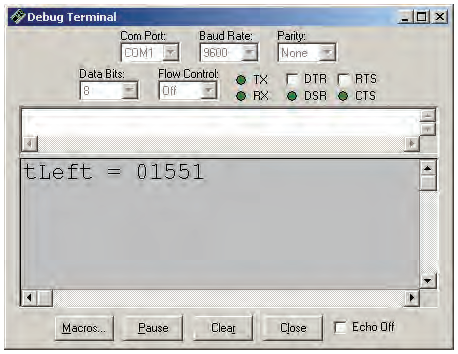
When the `RCTIME` command changes the pin direction from output to input, it stops charging the capacitor and becomes invisible to the circuit. As soon as that happens, the capacitor's charge starts draining through the phototransistor. As an input, the I/O pin can sense whether the circuit's voltage is above or below 1.4 V.

The RC in `RCTIME` stands for resistor-capacitor, and the **`RCTIME`** command's most common use is with sensors that vary with either resistance or capacitance. For an example of using this command to measure the position of a dial that controls resistance, see *What's a Microcontroller*?, Chapter 5. It's a free download from www.parallax.com/go/WAM.

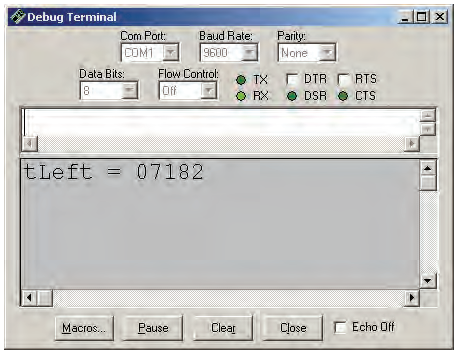
Test the Phototransistor Circuit

The TestP6LightSense.bs2 example program performs the three steps on the QT circuit connected to P6 in Figure 6-11 and displays a time measurements that represent the light level incident on the phototransistor's base terminal. The QT circuit connected to P6 is the Boe-Bot's left light sensor, and the Debug Terminal will display the decay time as **`tLeft`**, which is both the name of the variable that stores the result and an abbreviation of time-left. The value it displays is the decay time, measured in 2 µs time increments. This value will decrease with brighter light and increase with less bright light, like in Figure 6-12.

Figure 6-12: Two Different Light Levels Measured by Boe-Bot Robot's Left Light Sensor



Normal Indoor Lighting



Shade Over Sensor

Is `tLeft` stuck at 0 or 1? A 0 could mean it's way too dark, and a 1 could mean it's way too bright. Either one could also indicate a wiring error, so double check your circuits too.

- ✓ These sensor circuits are designed for indoor lighting. Make sure no direct sunlight is shining in through the windows. If there is, close the blinds.
- ✓ Enter and run TestP6LightSense.bs2.
- ✓ Make a note of the value displayed in the Debug Terminal.
- ✓ Use your hand or a book to cast a shadow over the phototransistor in the circuit connected to P6.
- ✓ Check the measurement in the Debug Terminal again. The value should be larger than the first one. Make a note of it too.
- ✓ Move the object casting the shadow closer to the top of the phototransistor. Try to make the shadow about twice as dark as the first shadow. Make a note of the measurement.
- ✓ Experiment with progressively darker shadows, even cupping your hand over the phototransistor. (If the light level gets low enough, the **`RCTIME`** command may exceed its maximum result value of 65535, in which case, the command will store a 0 in the **`tLeft`** variable, and the Debug Terminal will display "tLeft = 00000.")

```
' Robotics with the Boe-Bot - TestP6LightSense.bs2
' Test Boe-Bot's left photoresistor circuit.

' {$STAMP BS2}                                ' Target module = BASIC Stamp 2
' {$PBASIC 2.5}                               ' Language = PBASIC 2.5

tLeft          VAR     Word                  ' Stores left sensor decay time

PAUSE 1000                                   ' Wait 1 s before any DEBUG

DO                                           ' Main loop

  HIGH 6                                     ' 1 Set P6 high to start charging
  PAUSE 1                                    ' 2 Wait for cap to charge
  RCTIME 6, 1, tLeft                         ' 3 P6->input, measure decay time

  DEBUG HOME, "tLeft = ", DEC5 tLeft         ' Display result
  PAUSE 100                                  ' Wait 0.1 seconds

LOOP                                         ' Repeat main loop
```

Your Turn – Test the Other Phototransistor Circuit

The Boe-Bot robot's other phototransistor circuit is connected to P3. Before modifying your program to test the other circuit, it's always best to save the working program as-is first.

- ✓ Save TestP6LightSense.bs2, then save a copy as TestP3LightSense.bs2.
- ✓ Change the ***Pin*** argument from 6 to 3 in the **`HIGH`** and **`RCTIME`** commands.
- ✓ Change the variable name from **`tLeft`** to **`tRight`** in the **`VAR`** declaration, and in the **`RCTIME`** and **`DEBUG`** commands.
- ✓ Test and fix any typos or bugs.
- ✓ Update the comments at the beginning of the program.
- ✓ Save your modified program, and then run it.

It would also be nice to have a third program that tests both phototransistor circuits. As before, save one of the working programs, and then save a copy of it under a new name, like maybe TestP6P3LightSense.bs2. This program will need two variable declarations, and two sets of **`HIGH`**-**`PAUSE`**-**`RCTIME`** commands in its main loop. The **`DEBUG`** commands can be condensed into one. It might look something like this:

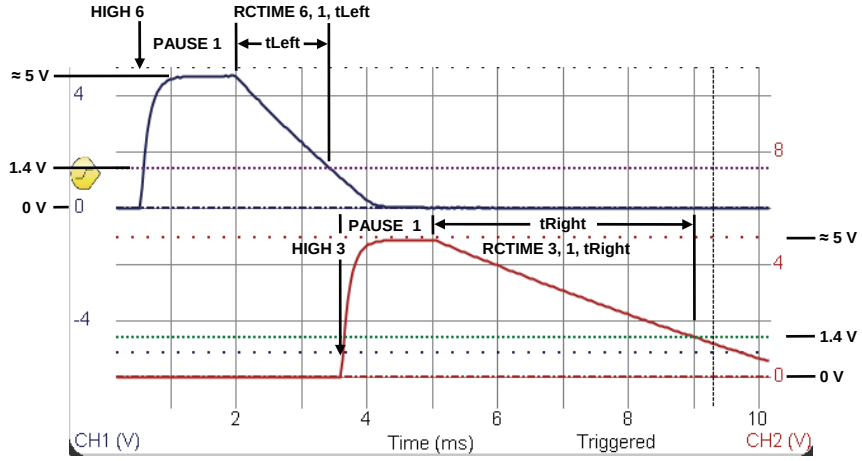
```
DEBUG HOME, "tLeft = ", DEC5 tLeft, "  ", "tRight = ", DEC5 tRight
```

- ✓ Try TestP6P3LightSense.bs2. It's in LightSensorExamples.zip, which is a free download from www.parallax.com/go/Boe-Bot.
- ✓ See if you can rotate the Boe-Bot and detect which is pointing toward the brightest light source in the room (lowest value) and which side is pointing away from it (higher value).

Optional Advanced Topic: Voltage Decay Graphs

Figure 6-13 shows the Boe-Bot robot's left and right QT circuit voltage responses while TestP6P3LightSense.bs2 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 show when each command in TestP6P3LightSense.bs2 executes, so that you can see how the voltage signals respond.

Figure 6-13: Oscilloscope View of Decay Times



The upper trace in Figure 6-13 plots the capacitor's voltage in the QT circuit connected to P6; that's the Boe-Bot's left light sensor circuit. In response to **`HIGH 6`**, the voltage rises from 0 V to almost 5 V between about 0.5 ms and 1 ms. The signal stays at around 5 V for the duration of **`PAUSE 1`**. Then, **`RCTIME`** causes the decay to start at about 2 ms into the plot. The **`RCTIME`** command measures the time it takes the voltage to decay to 1.4 V and stores it in the **`tLeft`** variable. In the plot, it looks like that decay took about 1.5 ms, so the **`tLeft`** variable should store something in the neighborhood of 750 since 750 × 2 µs = 1.5 ms.

The lower trace in Figure 6-13 plots the other QT circuit's capacitor voltage—the P3 sensor on the Boe-Bot's right side. This measurement starts after the left side P6 measurement is done. The voltage varies in a manner similar to the upper trace, except the decay time takes quite a bit longer, about 5 ms, and we would expect to see **`tRight`** store a value in the 2500 neighborhood. This larger value corresponds to a slower decay, which in turn corresponds to a lower light level.

Take your own oscilloscope measurements. You can measure and learn more about all the signals in this chapter with the *Understanding Signals with the PropScope* book and kit. To find out more, go to www.parallax.com/go/PropScope.

ACTIVITY #3: LIGHT SENSITIVITY ADJUSTMENT

If these **RCTIME** light measurements are going to be used while the Boe-Bot is roaming, they will have to share the BASIC Stamp's processing time with **PULSOUT** commands for servo control. There's a 20 ms time window between each pair of **PULSOUT** commands for **RCTIME** commands. Although 25 or 30 ms between servo pulses might not cause any noticeable difference, delays of more than 50 ms or so will cause noticeable problems, and larger delays will cause the servos to just twitch periodically instead of rotate.

A pair of phototransistor measurements in a really dark area might measure 50,000 each. For both measurements, that would be 100,000 × 2 µs = 400 ms. All the Boe-Bot's servos would do with this delay between servo control pulses is twitch every 0.4 seconds.

In this activity, you will try a technique that can be used to reduce decay time measurements in darker rooms. You will also test the effects of reduced light on servo performance using both measurement techniques.

Fix the Problem by Charging the Capacitor to a Lower Voltage with PWM

How can a program make the measurements take less time? By charging the capacitors to lower voltages before starting the decay measurements, the program can reduce the time the decays take to reach 1.4 V. The PBASIC language has a command called **PWM** that you can use to make the BASIC Stamp set the starting voltage across the capacitor to a lower value. This command gives you 256 voltage levels to choose from in the 0 to 4.98 V range. The **PWM** command's syntax is:

PWM *Pin*, *Duty*, *Duration*

The **PWM** command applies a rapid sequence of high/low signals to the I/O ***Pin*** for a certain ***Duration*** in ms. The ***Duty*** is the number of 256ths the time the signal is high, and it determines the number of 256ths of 5 V the capacitor gets charged to. For example, the command **PWM 6, 128, 1** sends a rapid sequence of high/low signals for 1 ms. They are high for 128/256ths of the time—that's half the time. So, it charges the capacitor to 128/256ths of 5 V. That's half of 5 V, which is 2.5 V.

PWM stands for Pulse Width Modulation. In Chapter 2, you studied pulse width modulation for servo control using **PULSOUT**. The **PWM** command makes the BASIC Stamp create another form of pulse width modulation. This signal is a more rapid sequence of pulses that's especially useful for setting voltage across a capacitor through a resistor. The proportion of high time to cycle time (high + low time) is what controls the capacitor voltage, and it is called duty cycle. The **PWM** command's ***Duty*** argument controls the PWM signals' duty cycle.

Given a **PWM** command, you can calculate the voltage it establishes across the capacitor by multiplying 5 V times the command's ***Duty*** argument and then dividing by 256:

Vcap = 5 V × Duty ÷ 256

Here are two examples:

```
PWM 6, 128, 1                   '  5 V × 128 ÷ 256 = 2.5 V.
PWM 6, 96, 1                    '  5 V × 96 ÷ 256 = 1.875 V.
```

Let's say you want to know what ***Duty*** value to use for a particular voltage. Just divide both sides of the equation by 5 V, and multiply both sides by 256, and the result is:

Duty = Vcap × 256 ÷ 5 V

A useful duty value to know would be Vcap = 1.4 V. Values below that wouldn't be any good for voltage decay time measurements.

Duty = 1.4 V × 256 ÷ 5 V = 71.68

So, a value of 72 would be the smallest useful ***Duty*** argument in **PWM 6, 72, 1**.

Why is the PWM command's *Duration* argument always 1 in these examples?

Because 1 ms is enough time to charge up a 0.1 µF capacitor through a 1 kΩ resistor. The general rule is that you need at least 5×R×C seconds to charge a capacitor. With a 1 kΩ resistor and 0.1 µF capacitor, the minimum would be 5 × 1000 × 0.0000001 = 0.0005 s = 0.5 ms. So, a 1 ms charge time is more than enough.

If the resistor or capacitor were larger, the **PWM** command's ***Duration*** argument might have to be larger. For example, if a 1 µF capacitor were used, the ***Duration*** argument would have to be at least 5 for a 5 ms of charging time because 5×R×C = 5 × 1000 × 0.000001 = 0.005 s = 5 ms. R×C is called the RC time constant, and is often abbreviated with the Greek letter tau τ. This letter is pronounced "taw."

Let's start by reducing the decay times to half of the values you measured in the previous activity. In practice, your program will need to reduce it by more to navigate in lower light levels, but this shows the first step in getting there. To reduce decay times by ½, you'll have to use the **PWM** command to charge the capacitors to half way between 1.4 V and 5 V. That corresponds to a **PWM** ***Duty*** value half way between 72 and 256, which is (72 + 256) ÷ 2 = 184. So, you can replace **HIGH 6** and **PAUSE 1** with **PWM 6, 184, 1** to reduce the decay times to half the values. Another way to think about this is that you are using the **PWM** command to make the sensors half as sensitive to light because the decay time measurements will take half as long for half the measured values.

- ✓ Enter, save and run HalfLightSensitivity.bs2.
- ✓ Try to make sure the ambient light is fairly close to the same level it was in the previous activity.
- ✓ Verify that the light measurements are about half of what they were with TestP6LightSense.bs2 from the previous activity. Precision is not important here. Don't worry if your measurements are not exactly one half of what they were; in the general neighborhood of one half is fine.
- ✓ Try changing the **PWM** command's ***Duty*** argument to 128 and verify that the measurements are now in the neighborhood of a quarter of the values from TestP6LightSense.bs2. Again, don't worry about being precise.


```
' Robotics with the Boe-Bot – HalfLightSensitivity.bs2
' Test Boe-Bot's photoresistor circuits with the PWM command cutting
' the phototransistor's light sensitivity in half.
' {$STAMP BS2}                                 ' Target module = BASIC Stamp 2
' {$PBASIC 2.5}                                ' Language = PBASIC 2.5

tLeft       VAR     Word                      ' Stores left sensor decay time
tRight      VAR     Word                      ' Stores right sensor decay time

PAUSE 1000                                    ' Wait 1 s before any DEBUG

DO                                            ' Main loop

  PWM 6, 184, 1                               ' Charge cap to 3.59 V
  RCTIME 6, 1,tLeft                           ' P6->input, measure decay time

  PWM 3, 184, 1                               ' Charge cap to 3.59 V
  RCTIME 3, 1,tRight                          ' P3->input, measure decay time

  DEBUG HOME, "tLeft = ", DEC5 tLeft, CR,     ' Display results
              "tRight = ", DEC5 tRight
  PAUSE 100                                   ' Wait 0.1 seconds

LOOP                                          ' Repeat main loop
```

One Light Sensor, Two Different Measurements

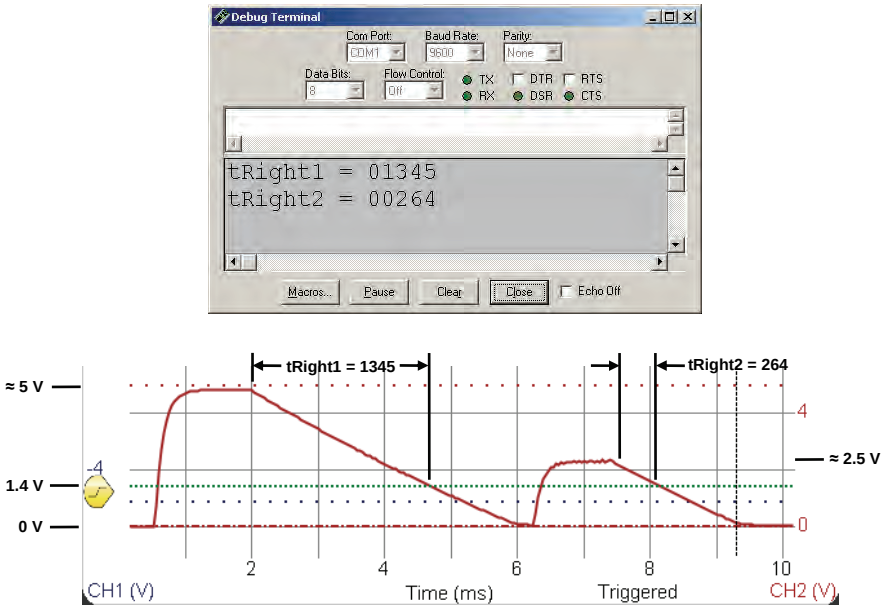
HighVsPwmInRctime.bs2 demonstrates how the decay measurement for the sensor takes less time when **PWM** charges it to a lower value.

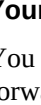
- ✓ Enter and run HighVsPwmInRctime.bs2, and observe the two measurements of the same light level in the Debug Terminal.
- ✓ Try varying the light level, the **tRight2** measurement should always be significantly smaller than **tRight1**.

```
' Robotics with the Boe-Bot - HighVsPwmInRctime.bs2
' Two decay measurements in a row.  The first uses the HIGH, PAUSE, RCTIME
' approach, and the second charges the capacitor to 2.5 V with PWM before
' RCTIME.

' {$STAMP BS2}                                 ' Target module = BASIC Stamp 2
' {$PBASIC 2.5}                                ' Language = PBASIC 2.5

tRight1      VAR     Word                      ' First right sensor decay time
tRight2      VAR     Word                      ' Second right sensor decay time

PAUSE 1000                                     ' Wait 1 s before any DEBUG

DO                                             ' Main loop

  HIGH 3                                       ' 1 Set P3 high to start charging
  PAUSE 1                                      ' 2 Wait for cap to charge
  RCTIME 3, 1, tRight1                         ' 3 P3->input, measure decay time

  PAUSE 1                                      ' Separate measurements by 1 ms

  PWM 3, 128, 1                                ' Charge P3 cap to 2.5 V
  RCTIME 3, 1,tRight2                          ' P3->input, measure decay time

  DEBUG HOME, "tRight1 = ", DEC5 tright1,      ' Display results
          CR, "tRight2 = ", DEC5 tRight2

  PAUSE 100                                    ' Wait 0.1 seconds

LOOP                                           ' Repeat main loop
```

Figure 6-14 shows both a Debug Terminal and an oscilloscope measurement of the two decays from HighVsPwmInRctime.bs2. In the oscilloscope trace, the first decay starts from 5 V because of **HIGH 3** and **PAUSE 1** before **RCTIME 3, 1, tRight1**. The second decay starts from 2.5 V because of **PWM 3, 128, 1** before **RCTIME 3, 1, tRight2**.

Figure 6-14: Debug Terminal and Oscilloscope Views of HighVsPwmInRctime.bs2





Why is `tRight2` more like 1/5th of `tRight1`? Isn't it supposed to be 1/4th? In the first decay, the I/O pin was output-high right up until the time **`RCTIME`** changes it to input. In the second decay, **`PWM`** changes the I/O pin to input when it is done, and then there is a brief delay between the end of the **`PWM`** command and the start of the **`RCTIME`** command. The voltage starts decaying at the end of the **`PWM`** command, so it is a little lower than 2.5 V by the time the **`RCTIME`** starts measuring the decay time.

This reduction in measurement value could be corrected with some testing, but won't matter because it will be the same for both right and left sensors. When Boe-Bot programs compare the two sensor values to determine which side is brighter or dimmer, both measurements will be lower by a small, fixed amount. So, if one sensor detects less light than the other, its measurement will still be larger, and that's what the program will need for navigation decisions.

Your Turn – Test Measurement Time's Impact on Servo Control

You can add a couple of **`PULSOUT`** commands that make the Boe-Bot go full speed forward to test the effect of measurement time on servo control. The first step would be a test to find out how low the light level has to get before the servos stop functioning properly with the **`HIGH-PAUSE-RCTIME`** approach. Then, use the **`PWM`** commands in place

of **HIGH** and **PAUSE**, with a ***Duty*** of 80, and test to find out how much darker it can get before the servo stop functioning properly.

- ✓ Run TestMaxDarkWithHighPause.bs2.
- ✓ Gradually increase the shade until the servos start twitching. (A shoebox would work well for this.)
- ✓ Repeat with TestMaxDarkWithPwm.bs2. The servos should still start twitching at some point, but it should be darker before it happens.

Figure 6-15 The Program on the Right Should Allow the Servos to Work in Lower Light

```
' TestMaxDarkWithHighPause.bs2

' {$STAMP BS2}
' {$PBASIC 2.5}

tLeft       VAR     Word
tRight      VAR     Word

PAUSE 1000
DEBUG "Program running... "

DO

  HIGH 6
  PAUSE 1
  RCTIME 6, 1,tLeft

  HIGH 3
  PAUSE 1
  RCTIME 3, 1,tRight

  PULSOUT 13, 850
  PULSOUT 12, 650

LOOP
```

```
' TestMaxDarkWithPwm.bs2

' {$STAMP BS2}
' {$PBASIC 2.5}

tLeft       VAR     Word
tRight      VAR     Word

PAUSE 1000
DEBUG "Program running... "

DO

  PWM 6, 80, 1
  RCTIME 6, 1,tLeft

  PWM 3, 80, 1
  RCTIME 3, 1,tRight

  PULSOUT 13, 850
  PULSOUT 12, 650

LOOP
```

ACTIVITY #4: LIGHT MEASUREMENTS FOR ROAMING

This activity features a program that automatically adjusts to the light conditions in the room and provides information about:

- How bright it is in the room
- Which of the two light sensors sees more shade
- How strong the light/dark contrast is between the two sensors

The Boe-Bot will be able to use this information for tasks like navigating toward or away from light.

The first example program looks pretty long, but don't worry. You'll be downloading it from www.parallax.com/go/Boe-Bot instead of hand-entering it.

Test LightSensorValues.bs2

LightSensorValues.bs2 utilizes several subroutines that condense the light measurements into two values: **light**, and **ndShade**. The **light** variable stores the ambient light level detected by the Boe-Bot. The **ndShade** variable stores a normalized, differential shade measurement. Normalized means that the measurements were fit to a certain scale, -500 to 500 in the case of **ndShade**. Differential means that the number corresponds to a difference between the two sensor measurements. In the case of **ndShade**, the value indicates the difference between the level of shade each sensor detects.

The program's **light** variable is useful for detecting overall light levels. The variable's scale is 1 to 324, with 1 being the darkest condition the system can measure and report and 324 being the brightest. This measurement is useful if a goal or waypoint in a navigation contest involves detecting when the robot passes under a bright light. In that situation, the **light** variable might store a larger value than elsewhere in the robot course, and the program could use an **IF…THEN** statement to detect that condition and take action.

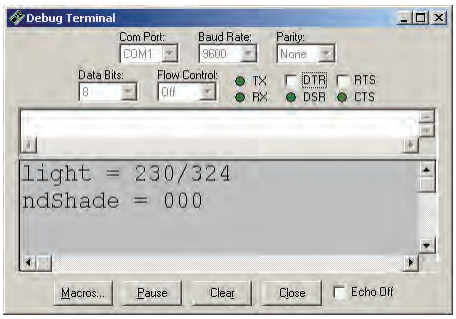
The program's **ndShade** variable indicates how much more shade one light sensor detects over the other. The variable's scale is -500 (shade much darker on left) to 500 (shade much darker on right). If the value of **ndShade** is 0, it means the light levels are about the same on both phototransistors. The measurement can be useful for code that makes the Boe-Bot roam either toward or away from light sources. For example, to make the Boe-Bot roam toward light, a routine simply has to make the Boe-Bot turn away if it detects shade on one side or the other.

Figure 6-16 shows examples of two different light/shade conditions measured with LightSensorsValues.bs2. The Debug Terminal on the left is an example of the Boe-Bot facing the main light source in a room. The **light** variable reports 230/324, which is in the normal range of indoor lighting. The **ndShade** variable reports 0, which means both phototransistors detect light levels that are very close to each other. The Debug Terminal on the right side of the figure shows a measurement with a shadow cast over the left light

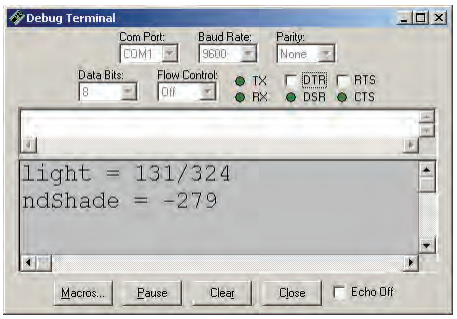
sensor. The value of **ndShade** is -279, which indicates shade over the left sensor, and the light value has dropped because shade over one sensor also reduces the total light measurement.

Figure 6-16: Example Shadow Tests with LightSeekingDisplay.bs2

Facing a Light Source



Shade over Boe-Bot's Left Light Sensor



- ✓ Download LightSensorExamples.zip from www.parallax.com/go/Boe-Bot.
- ✓ Make sure there is no direct sunlight streaming in nearby windows. Indoor lighting is good, but direct sunlight will blind the sensors.
- ✓ Unzip to a folder, and then open LightSensorValues.bs2.
- ✓ Open the program with your BASIC Stamp Editor and load it into the BASIC Stamp.
- ✓ For best results, adjust ambient lighting in the room so that the **`light`** variable is in the 125 to 275 range with no shade over the sensors.
- ✓ Verify that when you cast shade over the Boe-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-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.
- ✓ Verify that the **`light`** variable drops with increased shade and rises with more light.

```
'-----[ Title ]-------------------------------------------------------------
' Robotics with the Boe-Bot - LightSensorValues.bs2
' Displays conditioned ambient light level and differential shade on a scale
' of -500 to 500.

' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

'-----[ Constants/Variables ]-----------------------------------------------

Negative       CON     1                   ' For negative numbers

' Application Variables
light          VAR     Word                ' Brightness/darkness indicator
ndShade        VAR     Word                ' Normalized differential shade

' Subroutine Variables
tLeft          VAR     Word                ' Stores left RCTIME measurement
tRight         VAR     Word                ' Stores right RCTIME measurement
n              VAR     Word                ' Numerator
d              VAR     Word                ' Denominator
q              VAR     Word                ' Quotient
sumDiff        VAR     Word                ' For sum and difference calcs
duty           VAR     Byte                ' PWM duty argument variable
i              VAR     Nib                 ' Index counting variable
temp           VAR     Nib                 ' Temp storage for calcs
sign           VAR     Bit                 ' Var.BIT15 = 1 if neg, 0 if pos

'-----[ Initialization ]----------------------------------------------------

FREQOUT 4, 2000, 3000                      ' Start beep

DEBUG CLS                                  ' Clear Debug Terminal

'-----[ Main Routine ]------------------------------------------------------

DO
                                           ' Main Loop.
  GOSUB Light_Shade_Info                   ' Get light & ndShade

  DEBUG HOME, "light = ", DEC3 light,      ' Display light & ndShade
    "/324", CLREOL, CR,
    "ndShade = ", SDEC3  ndShade, CLREOL

  PAUSE 100                                ' Delay 0.1 seconds

LOOP                                       ' Repeat main loop

'-----[ Subroutine - Light_Shade_Info ]-------------------------------------
' Uses tLeft and tRight (RCTIME measurements) and pwm var to calculate:
'   o light   - Ambient light level on a scale of 0 to 324
'   o ndShade - Normalized differential shade on a scale of -500 to + 500
```

```
'               (-500 -> dark shade over left, 0 -> equal shade,
'                +500 -> dark shade over right)

Light_Shade_Info:                           ' Subroutine label
  GOSUB Light_Sensors                       ' Get raw RC light measurements
  sumdiff = (tLeft + tRight) MAX 65535      ' Start light level with sum
  IF duty <= 70 THEN                        ' If duty at min
    light=duty-(sumdiff/905) MIN 1          ' Find how much darker
    IF sumdiff = 0 THEN light = 0           ' If timeout, max darkness
  ELSEIF duty = 255 THEN                    ' If duty at max
    light=duty+((1800-(sumdiff))/26)        ' Find how much brighter
  ELSE                                      ' If duty in range
    light = duty                            ' light = duty
  ENDIF                                     ' Done with light level
  GOSUB Duty_Auto_Adjust                    ' Adjust PWM duty for next loop
  n = tLeft                                 ' Set up tLeft/(tLeft+tRight)
  d = tLeft + tRight
  GOSUB Fraction_Thousandths                ' Divide (returns thousandths)
  ndShade = 500-q                           ' Normalized differential shade
  RETURN                                    ' Return from subroutine

'-----[ Subroutine - Light_Sensors ]-----------------------------------------
' Measure P6 and P3 light sensor circuits.  Duty variable must be in 70...255.
' Stores results in tLeft and tRight.

Light_Sensors:                              ' Subroutine label
  PWM 6, duty, 1                            ' Charge cap in P6 circuit
  RCTIME 6, 1, tLeft                        ' Measure P6 decay
  PWM 3, duty, 1                            ' Charge cap in P3 circuit
  RCTIME 3, 1, tRight                       ' Measure decay
  RETURN                                    ' Return from subroutine

'-----[ Subroutine - Duty_Auto_Adjust ]--------------------------------------
' Adjust duty variable to keep tLeft + tRight in the 1800 to 2200 range.
' Requires sumdiff word variable for calculations.

Duty_Auto_Adjust:                           ' Subroutine label
  sumDiff = (tLeft + tRight) MAX 4000       ' Limit max ambient value
  IF sumDiff = 0 THEN sumDiff = 4000        ' If 0 (timeout) then 4000
  IF (sumDiff<=1800) OR (sumDiff>=2200) THEN ' If outside 1800 to 2200
    sumDiff = 2000 - sumDiff                ' Find excursion from target val
    sign = sumDiff.BIT15                    ' Pos/neg if .BIT15 = 0/1
    sumDiff = ABS(sumDiff) / 10             ' Max sumDiff will be +/- 10
    sumDiff = sumDiff MAX ((duty-68)/2)     ' Reduce adjustment increments
    sumDiff = sumDiff MAX ((257-duty)/2)    '  near ends of the range
    IF sign=NEGATIVE THEN sumDiff=-sumDiff  ' Restore sign
    duty = duty + sumDiff MIN 70 MAX 255    ' Limit duty to 70 to 255
  ENDIF                                     ' End of if outside 1800 to 2200
  RETURN                                    ' Return from subroutine
```

```
'-----[ Subroutine - Fraction_Thousandths ]-------------------------------------
' Calculate q = n/d as a number of thousandths.
' n and d should be unsigned and n < d.  Requires Nib size temp & i variables.

Fraction_Thousandths:                          ' Subroutine label
  q = 0                                        ' Clear quotient
  IF n > 6500 THEN                             ' If n > 6500
    temp = n / 6500                            '   scale n into 0..6500
    n = n / temp
    d = d / temp                               '   scale d with n
  ENDIF
  FOR i = 0 TO 3                               ' Long division ten thousandths
    n = n // d * 10                            ' Multiply remainder by 10
    q = q * 10 + (n/d)                         ' Add next digit to quotient
  NEXT
  IF q//10>=5 THEN q=q/10+1 ELSE q=q/10        ' Round q to nearest thousandth
  RETURN                                       ' Return from subroutine
```


Optional: How LightSensorValues.bs2 Works

The phototransistor circuits and **RCTIME** measurements pose two problems for Boe-Bot navigation. First, an **RCTIME** measurement for a shadow in a darker room will be a larger value than the same object casting the same shadow in a brighter room. Second, in darker rooms, the **RCTIME** measurements can end up taking a lot longer than the 20 ms of free time a program has between servo pulses.

The Main Routine in LightSensorValues.bs2 doesn't have to worry about either of those problems because the **Light_Shade_Info** subroutine solves them. The Main Routine just makes a single call to the **Light_Shade_Info** subroutine, and then checks the values of the **light** and **ndShade** variables for the two values it needs for navigation with a pair of light sensors. Again, the **light** variable indicates the overall light level on a scale of 0 to 324, and the **ndShade** variable indicates the light/shade difference between sensors on a scale of -500 to 500.

More detail about the subroutines: This section only focuses on what the subroutines do, not how they do it. Some more advanced activities that chronicle the development of the subroutines are available for download from www.parallax.com/go/Boe-Bot. Look for the Advanced Light Sensing section.

The **Light_Shade_Info** subroutine starts by calling the **Light_Sensors** subroutine to get the **tLeft** and **tRight** variables that store the **RCTIME** measurements. Notice that the **PWM** commands in the **Light_Sensors** subroutine rely on a variable named **duty** to set their sensitivity, which in turn controls how long the commands take to get their light

measurements. The program has a subroutine named **Duty_Auto_Adjust** that automatically adjusts the **duty** variable to help prevent rooms that are too dark from disabling the Boe-Bot's servos and rooms that are bright from blinding the light sensors.

After calling the **Light_Sensors** subroutine, the **Light_Shade_Info** subroutine does some math on **tLeft**, **tRight**, and the **duty** variable to calculate the **light** variable's value, which again indicates the overall light level. Next, it calls the **Duty_Auto_Adjust** subroutine, which adjusts the **duty** variable to try to keep the sum of the **RCTIME** measurements in the 1800 to 2200 range. Really dark rooms will still cause the servos to make the wheels twitch instead of turn, and direct sunlight will still blind the Boe-Bot, but **Duty_Auto_Adjust** significantly extends the range of light conditions that the Boe-Bot can automatically adjust to and navigate in.

Next, the **Light_Shade_Info** subroutine normalizes the difference between the two sensors by calculating how much of the total light (measured by both sensors) a single sensor sees. It does that by solving this equation:

$$ndShade = 500 - \left(1000 \times \frac{tLeft}{tLeft + tRight} \right)$$

This equation solves the problem of shade having different values in rooms with different light levels. It simply divides one measurement into the sum of both measurements for a fractional result that could range from 0 to 1. It then multiplies this by 1000 for a result that could range from 0 to 1000. It then subtracts all that from 500, for the **ndShade** variable value, which ranges from -500 to 500.

Let's say that **tLeft** is 1500 and **tRight** is 500. That means there's shade over the Boe-Bot's left light sensor. If you plug the values into the equation, the result will be -250. Now, in a darker room, that same shade condition might cause **tLeft** to be 3600 and **tRight** to be 1200. Those values still result in an **ndShade** value of -250.

- ✓ Use the **ndShade** equation to calculate both pairs of values discussed in the paragraph above.

You might have also noticed a new and different feature in the Constants/Variables section: **Negative CON 1**. This is a constant declaration, and it allows you to use a name in place of a number in your program. Instead of using the number 1 at a certain point in the program to check to find out if a number is negative, the program uses the

Negative constant instead. So, down in the **Duty_Auto_Adjust** subroutine, the statement **IF sign=Negative THEN sumDiff = -sumDiff** checks to find out if the **sign** variable contains a 1, indicating that a value tested as negative earlier in the subroutine. This line would still work if it was rewritten **IF sign=1 THEN sumDiff = -sumDiff**.

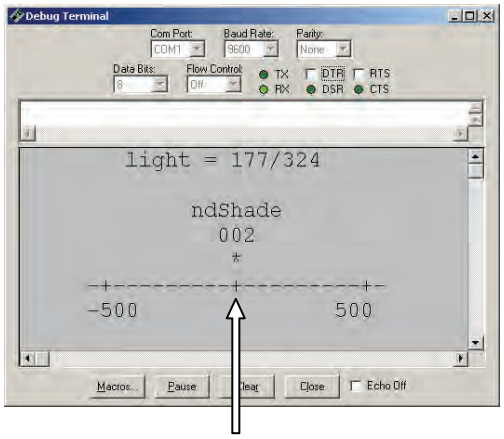
Constants can be useful for helping commands with numbers in them be more self explanatory, and are also useful if you have a number that is used several places in a program. By updating one **CON** directive, all the code that uses the constant's name will use the updated value. Chapter 8 will utilize this feature of constants for calibrating a program that makes the Boe-Bot follow objects within a certain range of its infrared object sensors.

Light Measurement Graphic Display

Figure 6-17 shows an example of a graphical display of the **ndShade** variable. The asterisk will be in the center of the -500 to 500 scale if the light or shade is the same over both sensors. If the shade is darker over the Boe-Bot's left sensor, the asterisk will position to the left in the scale. If it's darker over the right, the asterisk will position toward the right. A larger shade/light contrast (like darker shade over one of the sensors) will result in the asterisk positioning further from the center.

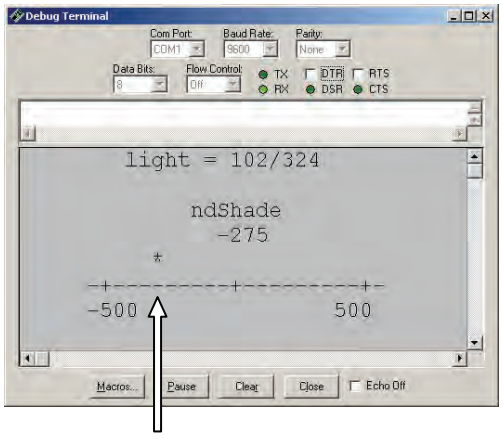
Figure 6-17: Graphic Display of ndShade Variable

Same light/shade on both sides



Asterisk indicator in center of scale

Shade darker over left side



Asterisk indicator about half way between center and far left

All you need for this display is some small modifications to LightSensorValues.bs2's Initialization and Main Routine sections. Below is an example. It makes use of some new **DEBUG** formatters, like **REP** and **CRSRX**.

The **REP** formatter repeats a character a certain number of times. So **DEBUG CLS, REP CR\5** clears the screen, and then prints 5 carriage returns, which sends the cursor down 5 lines.

The **CRSRX** formatter positions the cursor a certain number of spaces to the right of the Debug Terminal's left margin. For example, **DEBUG HOME, CRSRX 8** sends the cursor to the Debug Terminal's top-left character position, then it moves the cursor eight spaces to the right.

CLREOL is another new formatter that erases everything to the right of the cursor on a given line. This can be useful when you don't necessarily know how many digits will be displayed. If a measurement displays fewer digits than the one before it, the **CLREOL** formatter erases any phantom digits that might be left over to the right.

```
' Excerpt from LightSensorDisplay.bs2

'-----[ Initialization ]----------------------------------------------------
FREQOUT 4, 2000, 3000                       ' Start beep

DEBUG CLS, REP CR\5,
      "     -+---------+---------+-",        ' Shade level graphical view
  CR, "     -500               500"

'-----[ Main Routine ]------------------------------------------------------
DO                                          ' Main Loop.

  GOSUB Light_Shade_Info                    ' Get light & ndShade
                                            ' Display
  DEBUG HOME, CRSRX, 8, "light = ",         '  light variable name at x=8
      DEC3 light, "/324",CR, CR,            '  light variable value
      CRSRX, 13, "ndShade", CR,             '  shade heading at x = 13
      CRSRX,15, SDEC3 ndShade, CLREOL, CR,  '  ndShade value at x = 15
      CLREOL, CRSRX,                        '  display asterisk at ndShade
      6+((ndShade+500)/50), "*"             '   x-position

  PAUSE 100                                 ' Delay 0.1 seconds

  LOOP                                      ' Repeat main loop
```

- ✓ LightSensorDisplay.bs2 was another example in LightSensorExamples.zip. Open it with the BASIC Stamp Editor.
- ✓ If you would prefer to save LightSensorValues.bs2 as LightSensorDisplay.bs2 and hand enter the changes, make sure to leave five spaces between the quotation marks and the first characters in each scale. For example, there are 5 spaces between the quotation marks and the first dash in **`CR, "     -500…`**
- ✓ Remember, for best results, make sure to adjust the area lighting so that the Debug Terminal displays light values in the 125 to 275 range with no shadows over the phototransistors.
- ✓ Run the program and try casting different levels of shade over each light sensor, and watch how the asterisk in Figure 6-17 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.

ACTIVITY #5: ROUTINE FOR ROAMING TOWARD LIGHT

One approach toward making the Boe-Bot roam toward light sources is to make it turn away from shade. You can even use the **`ndShade`** variable to make the Boe-Bot turn more or less when the contrast between the light detected on each side is more or less. First, we need a couple variables to store pulse duration variables for the servos.

```
' Application Variables
pulseLeft      VAR     Word
pulseRight     VAR     Word
```

Next, we need some code to set those pulse values. The code below sets **`pulseLeft`** and **`pulseRight`** to keep the wheel under shade going full speed and slow down or reverse the other wheel. When the contrast between light and shade measurements is small, the wheel that's not under shade only slows down somewhat for a gradual turn. When the contrast is larger, the wheel on the other side from the shade may slow down more, or even start turning backwards so that the Boe-Bot executes a sharper turn away from that darker shadow.

```
' Navigation Routine
IF (ndShade + 500) > 500 THEN
  pulseLeft = 900 - ndShade MIN 650 MAX 850
  pulseRight = 650
ELSE
  pulseLeft = 850
  pulseRight= 600 - ndShade MIN 650 MAX 850
ENDIF
```

The routine that sets **`pulseLeft`** and **`pulseRight`** values starts by deciding if the shade is over the right or left sensor, by comparing (**`ndShade + 500`**) to **`500`**. The > (greater than), >= (greater than or equal to), < (less than), and <= (less than or equal to) operators only compare two positive numbers. Since the smallest **`ndShade`** could be is -500, the code in the **`IF`** condition adds 500 to **`ndShade`** and then compares it to 500. It's the PBASIC equivalent to **`IF ndShade > 0`**.

Let's say that **`ndShade`** is 125, which means there's definitely some shade over the right light sensor. **`IF ndShade + 500 > 500 THEN`** checks if 625 is greater than 500, which it is. Figure 6-18 shows what happens next as the code slows down the left wheel with **`pulseLeft = 900 – ndShade MIN 650 MAX 850`**, and sets the right wheel to full speed forward with **`pulseRight = 650`**. Since **`ndShade`** is 125 in this example, 900 – 125 = 775, which would cause a **`PULSOUT`** command to slow down the left wheel.

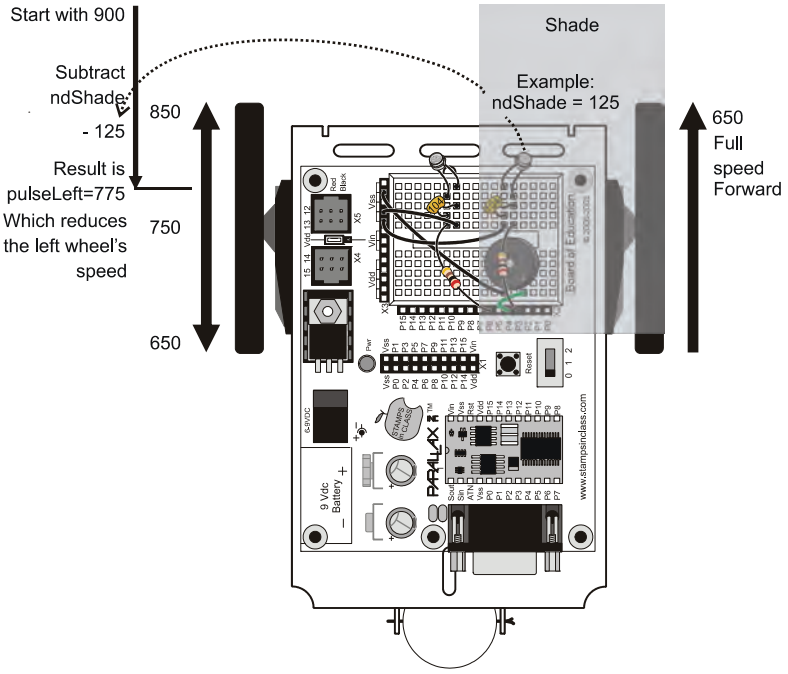


Figure 6-18
LightSeekingBoe-Bot.bs2's Reaction to Shade on Right

In this example, an ndShade measurement of 125 gets subtracted from 900, and the result of 775 slows down the left servo's speed.

If **ndShade** is larger, like maybe 190, which means the shade over the right sensor is darker, **pulseLeft** ends up with a value of 710, which would make the left wheel turn backwards for a much sharper turn. For values of **ndShade** that are greater than 250, the expression **900 - ndShade** might result in values smaller than 650. Likewise, for values of **ndShade** between 1 and 49, the expression might result in values above 850. So, the code uses **MIN** and **MAX** operators to keep the result in the 650 to 850 range even though **900 - ndShade** might have intermediate results outside that range.

The **MIN** operator takes a result below the specified value and increases it to that value, but leaves results above the **MIN** value alone. So, if the result of **600 - ndShade** is anything below 650, the **MIN** operator stores 650 in **pulseLeft**.

For example, if **ndShade** were 350, the intermediate result of **900 - ndShade** would be 550, but **MIN 650** would change that to 650. Similarly, the **MAX** operator takes a result above the specified value and decreases it to that value, but leaves results below the **MAX** value alone. So, even though values from 0 to 49 would yield intermediate **900 - ndShade** results in the 900 to 851 range, the **MAX 850** part of the expression sets any result in that range to 850.

For **ndShade** values of zero or less, it means shade is over the left sensor, and the right wheel needs to slow down. The code in the **ELSE** block does that by setting the left wheel to full speed with **pulseLeft = 850** to make the Boe-Bot's left wheel go full speed forward, and **pulseRight = 600 - ndShade MIN 650 MAX 850** to slow down or even reverse the direction of the Boe-Bot's right wheel, depending on how dark the shade is over the left light sensor.

Test Navigation Routine with Debug Terminal

Figure 6-19 shows some Debug Terminal display examples from the next example program, LightSeekingDisplay.bs2. Checklist instructions will prompt you to run the program next, but first, just take a look at the Debug Terminal displays in the figure.

These screen captures demonstrate how the navigation routine adjusts the **`pulseLeft`** and **`pulseRight`** variables in response to different **`ndShade`** values. The program makes the Debug Terminal display a top-view of the Boe-Bot with **`pulseLeft`** and **`pulseRight`** labels and their values next to each wheel. The program also positions the left > and right < wheel speed indicators to show how fast and in which direction each wheel would be turning.

For example, in the upper-left Debug Terminal, both wheel speed indicators are level with the forward label, which means the Boe-Bot would be going full speed forward.

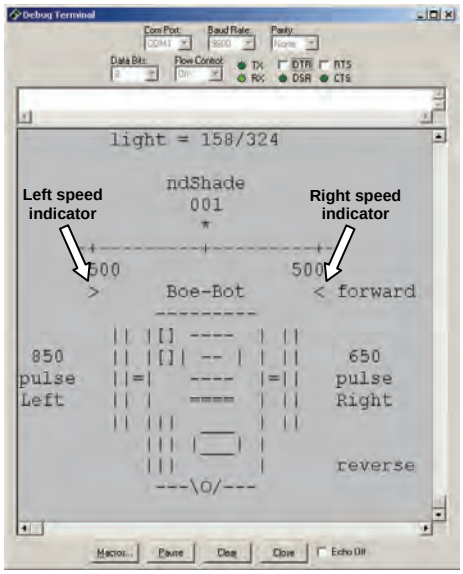
In the upper-right Debug Terminal, the right speed indicator is half way between the forward and backward labels which means that wheel would stop.

In the lower-left Debug Terminal, the left wheel speed indicator is level with the reverse label, which means the left wheel would be turning full speed in reverse.

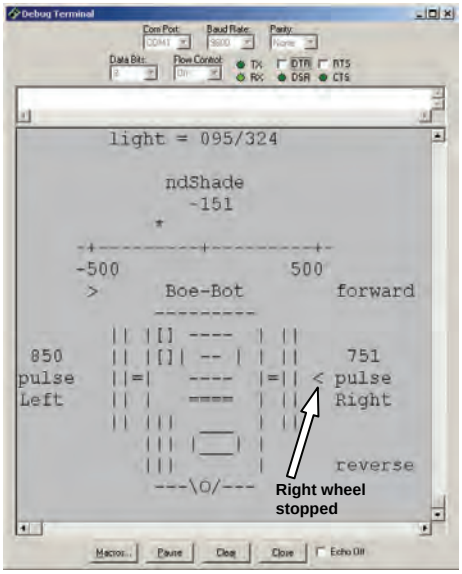
In the lower-right, the **`light`** variable is lower. Since **`ndShade`** is close to zero, the level of shade is about the same, so there must be shade over both sensors. Since the Boe-Bot only responds to differences in shade, the same shade over both sensors means that both wheels would be turning full speed forward again.

Figure 6-19: Shade and Wheel Speed Indicator Examples

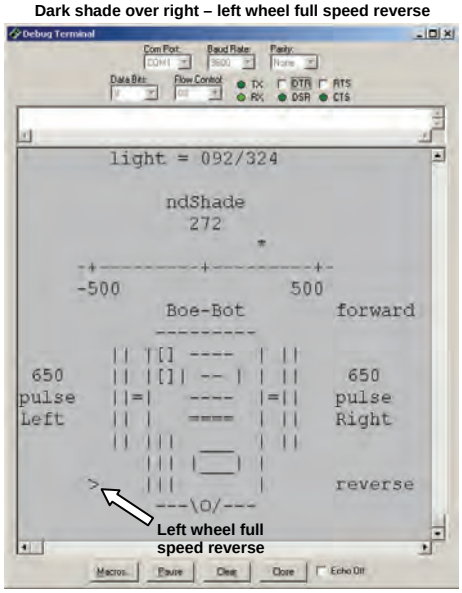
Equal light – full speed forward



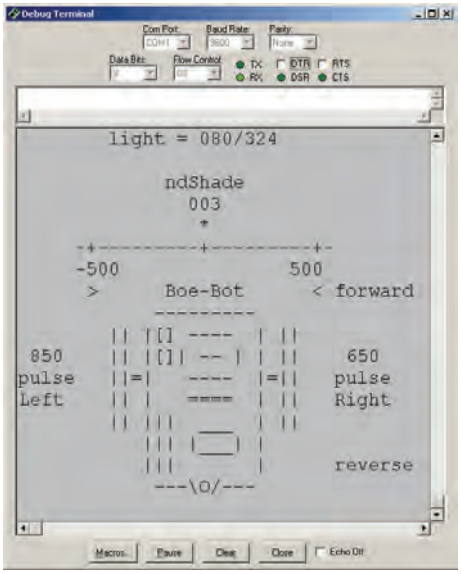
Shade over left – slow down right wheel



Dark shade over right – left wheel full speed reverse



Equal shade – back to full speed forward



6

LightSeekingDisplay.bs2 is another example from the LightSensorExamples.zip archive. It expands on LightSensorDisplay.bs2 with these features:

- Word-size variable declarations for **`pulseLeft`** and **`pulseRight`**
- A **`DEBUG`** command that displays a top view of the Boe-Bot
- The light seeking routine **`IF…THEN…ELSE…ENDIF`** code block
- Debug commands that display the **`pulseLeft`** and **`pulseRight`** variable values next to each wheel.
- **`DEBUG`** commands that position the left wheel speed indicator > and right wheel speed indicator < to show the speed and direction of each wheel.

LightSeekingDisplay.bs2 is a great program for observing the **`pulseLeft`** and **`pulseRight`** variable responses to shadows over each sensor and how those values affect wheel speed.

- ✓ Open LightSeekingDisplay.bs2 with the BASIC Stamp Editor and download it to the BASIC Stamp.
- ✓ Again, for best results, adjust ambient lighting in the room so that the Debug Terminal displays a **`light`** variable value in the 125 to 275 range with no shade over the sensors.
- ✓ Experiment with more and less shade over each sensor, and pay careful attention to how that affects the **`ndShade`** value, which in turn affects the **`pulseLeft`** and **`pulseRight`** variables and the wheel speeds that they would set if used in **`PULSOUT`** commands.

```
' Excerpts from LightSeekingDisplay.bs2

'   ...  (three dots indicate code omitted)

' Application Variables
pulseLeft      VAR     Word                  ' Left servo pulse duration
pulseRight     VAR     Word                  ' Right servo pulse duration

'   ...
```

```
'-----[ Initialization ]---------------------------------------------------
FREQOUT 4, 2000, 3000                       ' Start beep

DEBUG CLS, REP CR\5,
      "     -+---------+---------+-",       ' Shade level graphical view
  CR, "     -500               500",
  CR, "             Boe-Bot        forward",
  CR, "            ---------       ",       ' Boe-Bot top view with pulse
  CR, "        || |[] ----  | ||   ",       '  labels
  CR, "        || |[]| -- | | ||   ",
  CR, "pulse   ||=|   ----  |=||   pulse ",
  CR, "Left    || |   ====  | ||   Right ",
  CR, "        || |||  ___  | ||   ",
  CR, "           ||| |___| |      ",
  CR, "           |||       |      reverse",
  CR, "            ---\O/---       "

'      0123456789    +10   0123456789    +30 ' Cursor positions to help if you
'          +0    0123456789   +20    01234567'  are hand entering the code.
'-----[ Main Routine ]-----------------------------------------------------
DO                                          ' Main Loop.

  GOSUB Light_Shade_Info                    ' Get light & ndShade
                                            ' Display
  DEBUG HOME, CRSRX, 8, "light = ",         '  light variable name at x=8
      DEC3 light, "/324",CR, CR,            '  light variable value
      CRSRX, 13, "ndShade", CR,             '  shade heading at x = 13
      CRSRX,15, SDEC3 ndShade, CLREOL, CR,  '  ndShade value at x = 15
      CLREOL, CRSRX,                        '  display asterisk at ndShade
      6+((ndShade+500)/50), "*"             '   x-position

  ' Navigation Routine
  IF (ndShade + 500) > 500 THEN             ' If more shade on right...
    pulseLeft = 900 - ndShade MIN 650 MAX 850' Slow left wheel w/ right shade
    pulseRight = 650                        ' Right wheel full speed forward
  ELSE                                      ' If more shade on left...
    pulseLeft = 850                         ' Left wheel full speed forward
    pulseRight= 600 - ndShade MIN 650 MAX 850' Slow right wheel w/ left shade
  ENDIF

  DEBUG CRSRXY, 1, 10, DEC3 pulseLeft,      ' Display pulse variable values
        CRSRX, 29, DEC3 pulseRight          '  above variable names

  FOR i = 7 TO 15                           ' Clear areas where > and <
    DEBUG CRSRXY, 6, i, " ", CRSRX, 26, " " '  wheel speed and direction
  NEXT                                      '  indicators might be placed

  DEBUG CRSRXY,6,15-((pulseLeft-650)/25),">" ' Place new wheel speed and
  DEBUG CRSRXY,26,7+((pulseRight-650)/25),"<"'  direction indicators

LOOP                                        ' Repeat main loop
' ...
```

LightSeekingDisplay.bs2 utilizes another **DEBUG** formatter, **CRSRXY**, to position the cursor to display each wheel speed indicator. **CRXRXY** should be followed by two numbers. For example, **DEBUG CRXRXY, 6, 11, ">"** would display the > character at six spaces from the Debug Terminal's left margin, and 11 carriage returns down from the top. The program actually uses an expression to set the number of carriage returns for positioning the cursor. For example, the command:

```
DEBUG CRSRXY,6,15-((pulseLeft-650)/25),">"
```

...positions the cursor 6 spaces from the Debug Terminal's far left, but it uses an expression to choose the number of carriage returns from the Debug Terminal's top line. Let's say that **pulseLeft** is 750. Then, the y position would be 11 because 15 – ((750 – 650)/25 = 15 – 4 = 11. So in that case, **CRSRXY** positions the cursor at 6, 11, and then prints the ">" character.

Your Turn – Save Lots of RAM

You might want to add other features to your Boe-Bot on top of light seeking, but that might be difficult if your application runs out of RAM just because you try to declare a few more variables. The left side of Figure 6-20 shows the problem, the application has slightly less than 2 words left. The right side of the figure shows how much space you can save by using a simple technique called variable aliasing.

- ✓ To see the RAM Map for LightSeekingDisplay.bs2, click the Memory Map button; it's just to the left of the Run button. You can also display it by clicking Run → Memory Map, or by pressing the CTRL + M keys. Your RAM Map should resemble the one on the left side of Figure 6-20.

According to the BASIC Stamp Help, an alias is "an alternative name for an existing variable."

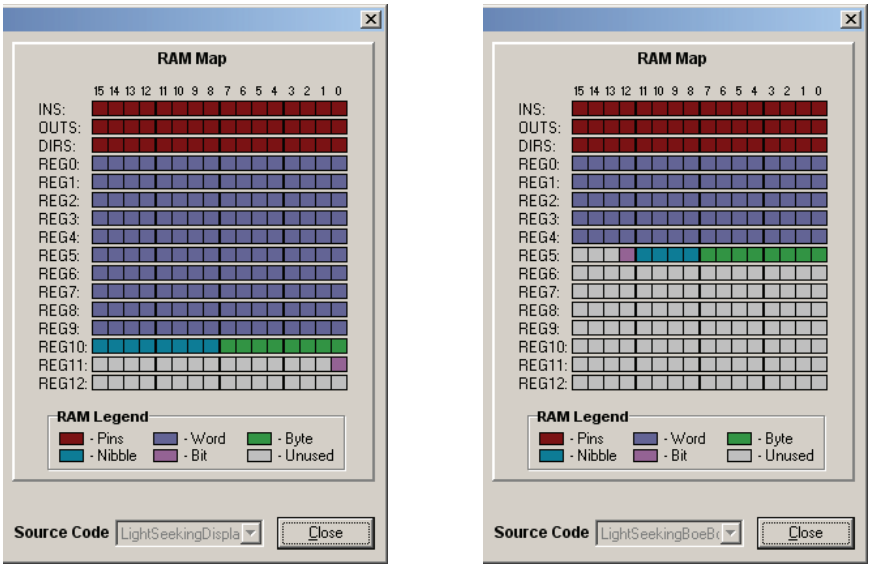
- ✓ In the BASIC Stamp Editor, click Help and select BASIC Stamp Help.
- ✓ Click PBASIC Language Reference and then click Variables to display the Variables page. Find the alias explanation, read it and examine the example variable declaration that uses aliases.

Not all of the variables in LightSeekingDisplay.bs2 are used all the time. For example, the program is done with **tLeft** and **tRight** after the **Light_Shade_Info** subroutine is done. Further, it never uses those two variables at the same time that it uses **pulseLeft**

and **pulseRight**. So **tLeft** can be declared as an alias of **pulseLeft** and **tRight** can be declared as an alias for **pulseRight**. Now, **pulseLeft** and **tLeft** use the same piece of memory, and likewise with **pulseRight** and **tRight**, and your application just recovered two words of RAM.

With the modifications in LightSeekingDisplayBetterRAM.bs2, the program reduces RAM usage from "almost all" to "less than half."

Figure 6-20: Variable Aliases Save almost Half of the BASIC Stamp's RAM



- ✓ Save LightSeekingDisplay.bs2 as LightSeekingDisplayBetterRAM.bs2.
- ✓ Update the variable declarations in LightSeekingDisplay.bs2 so that they match the right side of Figure 6-21.
- ✓ Recheck your Memory Map. It should now resemble the one on the right side of Figure 6-20.

Figure 6-21 Saving Space with Variable Aliases

```
' Application Variables
pulseLeft      VAR     Word
pulseRight     VAR     Word
light          VAR     Word
ndShade        VAR     Word

' Subroutine Variables
tLeft          VAR     Word
tRight         VAR     Word
n              VAR     Word
d              VAR     Word
q              VAR     Word
sumDiff        VAR     Word
duty           VAR     Byte
i              VAR     Nib
temp           VAR     Nib
sign           VAR     Bit
```

```
' Application Variables
pulseLeft      VAR     Word
pulseRight     VAR     Word
light          VAR     Word
ndShade        VAR     Word

' Subroutine Variables
tLeft          VAR     pulseLeft
tRight         VAR     pulseRight
n              VAR     tLeft
d              VAR     Word
q              VAR     ndShade
sumDiff        VAR     d
duty           VAR     Byte
i              VAR     Nib
temp           VAR     i
sign           VAR     Bit
```

CAUTION: Be careful how you use variable aliases. If the program needs two variables at the same time, one variable cannot be an alias for the other. Likewise, if the program needs to check the previous value of a variable in the next iteration of a loop, giving it an alias and using it for another purpose would erase a value your program needs later.

For example, if you tried to make **`pulseLeft`** an alias for **`pulseRight`**, both your servo speeds would always be the same. They could not be the two independent values your code needs for servo control.

ACTIVITY #6: TEST NAVIGATION ROUTINE WITH THE BOE-BOT

This code excerpt from LightSeekingBoeBot.bs2 has a navigation-only version of the Initialization and Main Routines from LightSeekingDisplayBetterRAM.bs2. All the **`DEBUG`** commands have been removed along with that **`100 ms PAUSE`** command. They all got replaced with **`PULSOUT`** commands that use the **`pulseLeft`** and **`pulseRight`** variables to control the servos.

```
'-----[ Initialization ]--------------------------------------------------------
FREQOUT 4, 2000, 3000                       ' Start beep
DEBUG "Program running..."                  ' Display program running message

'-----[ Main Routine ]----------------------------------------------------------
DO                                          ' Main Loop.

  GOSUB Light_Shade_Info                    ' Get light & ndShade

  ' Navigation Routine
  IF (ndShade + 500) > 500 THEN             ' If more shade on right...
    pulseLeft = 900 - ndShade MIN 650 MAX 850' Slow left wheel w/ right shade
    pulseRight = 650                        ' Right wheel full speed forward
  ELSE                                      ' If more shade on left...
    pulseLeft = 850                         ' Left wheel full speed forward
    pulseRight= 600 - ndShade MIN 650 MAX 850' Slow right wheel w/ left shade
  ENDIF

  PULSOUT 13, pulseLeft                     ' Left servo control pulse
  PULSOUT 12, pulseRight                    ' Right servo control pulse

LOOP                                        ' Repeat main loop
```


- ✓ Open LightSeekingBoeBot.bs2 with the BASIC Stamp Editor and load it into your BASIC Stamp.
- ✓ Or, save LightSeekingDisplayBetterRAM.bs2 as LightSeekingBoeBot.bs2 and update the Initialization and Main Routine so that they match the code snippet above.
- ✓ Run the program.
- ✓ If you have a Board of Education, set the 3-position switch to 2 after you have disconnected the Boe-Bot from its programming cable and set it where you want it to start roaming.
- ✓ Your Boe-Bot can now roam toward the light.
- ✓ Try casting shadows over your Boe-Bot's light sensors as it roams. It should turn away from the shadows.
- ✓ Try sending your Boe-Bot toward a dark shadow cast by a desk. Make sure its approach is at an angle instead of making it drive straight into the shadow. It should veer away.
- ✓ Try taking the Boe-Bot into a low light room with bright light streaming in the doorway. Can it find its way out?

Troubleshooting

If the Boe-Bot robot seems a little less sensitive to light on one side, try correcting it by following the instructions in the next section (Your Turn – Light/Shade Sensitivity Adjustments). The same applies if you want to the Boe-Bot to either be more or less sensitive to shade.

If the Boe-Bot does not respond to shadows by turning away from them, or if it turns in place instead of roaming, follow these steps:

- If you hand-entered your code, try the LightSeekingBoeBot.bs2 code example in LightSensorExamples.zip, a free download from www.parallax.com/go/Boe-Bot. This will rule out coding errors before taking a closer look at your circuit.
- If code from the Parallax web site doesn't fix the problem, you'll need to check the circuit next. Start by carefully verifying all your circuit connections against the schematic and wiring diagram in Activity #2. Double check the resistor color codes (brown-black-red), capacitor numbers (104), the phototransistor pin lengths, and make sure that all the leads are connected as shown in the wiring diagram. Also make sure to verify that you selected phototransistors and not infrared LEDs with the help of Figure 6-8 on page 180. Verify that the phototransistors are pointing upwards and outwards, like in Figure 6-10 on page 183. Sometimes, pointing them a little further outwards improves responses to shade. As you adjust the directions the phototransistors point, make sure that their leads do not touch each other. Repeat the Debug Terminal tests, and make sure to do it for both sensors. Each one should respond similarly to light and shade.
- Next, repeat the tests in Activity #5. Use the Debug Terminal to verify that the light levels are in the 125 to 275 range. Also, shade over a given sensor should slow down the servo on the other side. Similar shade over the other sensor should result in a similar motor speed adjustment of the other wheel. Also verify that the same light or shade level over both sensors results in full-speed-forward.
- If the Debug Terminal displays a value of **ndShade** that's way off from zero, even when the sensors see about the same light level, there are some additional tests you can try in the Phototransistor Matching activity. It's part of the advanced light sensing activities available from www.parallax.com/go/Boe-Bot. If the Debug Terminal instead indicates that the sensors and servos are both responding to shade and light correctly, try LightSeekingBoeBot.bs2 again. If it still doesn't respond correctly to shadows, it's time to check your servo motors. Repeat Chapter 2, Activity #6. For a closer look, make the graphs in Chapter 3, Activity #4 for both servos.

For more troubleshooting help, try the www.parallax.com/support resources.

Your Turn – Light/Shade Sensitivity Adjustments

You can change the 900 and 600 in these lines to make the Boe-Bot more or less sensitive to shade:

```
pulseLeft = 900 - ndShade MIN 650 MAX 850
```

...and:

```
pulseRight= 600 - ndShade MIN 650 MAX 850
```

For example, you can increase the Boe-Bot's light/shade sensitivity by changing 900 to 875 and changing 600 to 625. You can make it even more sensitive by changing the 900 to 850 and the 600 to 650. Likewise, you could make the whole system less sensitive by changing the 900 to 925, and the 600 to 575, and so on…

- ✓ Try it.

You can also adjust one of the values to make either the left or right sensor more sensitive. Changing 900 to another value would change the Boe-Bot's sensitivity to shade on the right, while changing 600 to another value would change the Boe-Bot's sensitivity to shadows on the right.

- ✓ Try that too.

Other things you can do with minimal adjustments to the Main Routine include:

- Following shade instead of light with **`ndShade = -ndShade`** right after the Main Routine's **`Light_Shade_Info`** subroutine call.
- Ending roaming under a bright light or in a dark cubby by detecting very bright or dark conditions with the **`light`** variable with an **`IF…THEN`** statement.
- Functioning as a light compass by remaining stationary and turning toward bright light sources.
- Incorporating whiskers into the roaming toward light so that the Boe-Bot can detect and navigate around objects in its way.

SUMMARY

A phototransistor is a light–controlled current valve. It lets more current through with brighter incident light and less current through with less bright light. This chapter utilized two different phototransistor circuits to detect light: a voltage output circuit and a charge transfer circuit.

The phototransistor voltage output circuit in this chapter was connected to an I/O pin set to input for a binary value that indicated bright or ambient light. When the phototransistor lets more current through, the voltage across the resistor is larger. When it lets less current through, the voltage across the resistor is smaller. By choosing the right resistor for the lighting conditions, the circuit can be monitored by an I/O pin because its voltage will go above 1.4 V in bright light, and below 1.4 V in ambient light. The I/O pin's input register stores a 1 when the voltage is above 1.4 V and a 0 when it's below 1.4 V.

A pair of charge transfer circuits were used for measuring differences in light intensity between the left and right phototransistor, and the Boe-Bot was programmed to detect and act on those differences. The charge transfer circuit consisted of a parallel capacitor and phototransistor connected to an I/O pin with a resistor. In this circuit, the BASIC Stamp used an I/O pin to charge the capacitor. Then, it switched the I/O pin to input and measured the time it took the capacitor's voltage to decay as it lost its charge through the phototransistor. This decay time measurement turns out to be smaller with bright light and larger in shade.

A **HIGH** command followed by a **PAUSE** can charge the capacitor, and then the **RCTIME** command changes the I/O pin to input and measures the time it takes for the capacitor's voltage to decay to 1.4 V as it loses its charge through the phototransistor. The measurement's time can be reduced by using the **PWM** command in place of **HIGH** and **PAUSE**. The **PWM** command can charge the capacitor to values less than 5 V before the **RCTIME** command, so the capacitor has fewer volts to decay before it reaches 1.4 V, and this takes less time. The time reduction helps keep the delay between **PULSOUT** commands from getting so large that it makes the servos twitch instead of turn.

Activity #4 through Activity #6 utilize a collection of subroutines that supply the Main Routine with values of overall light levels along with the light/shade contrast between the two sensors. This light/shade contrast between the sensors is called a differential measurement, and the subroutines also normalize the measurement to a scale of -500 to 500. The actual decay measurements may vary depending on ambient light levels, so the normalized values keep these measurements on a scale that is useful for servo control and Boe-Bot navigation toward light sources.

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_{P6} in Figure 6-6 respond if the light gets brighter?
6. What does the phototransistor in Figure 6-6 do that causes V_{P6} to increase or decrease?
7. How can the circuit in Figure 6-6 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_{P6} if I = 1 mA in Figure 6-6.
2. Calculate the current through the resistor if V_{P6} in Figure 6-6 is 4.5 V.
3. Assume that the threshold between light and dark needed for your application occurs when V_{P6} = 2.8 V. Calculate the resistor value you would need for the BASIC Stamp to detect this threshold.
4. Calculate the value of a capacitor that has been stamped 105.
5. Write an **`RCTIME`** command that measures decay time with I/O pin P7 and stores the result in a variable named **`tDecay`**.
6. Write a **`PWM`** command that charges the capacitor in Figure 6-11 to about 1.625 V to prepare the circuit for a decay measurement.
7. Calculate what the **`ndShade`** measurement would be if the BASIC Stamp measures decay values of 1001 on both sides.
8. Write a **`DEBUG`** command that displays fifty equal sign characters.

9. Write a **DEBUG** command that positions the character "#" eight spaces to the right of the Debug Terminal's left margin and 10 carriage returns down from the top.

Projects

1. In Activity #1, the circuit along with the example code in the Your Turn section made the Boe-Bot stop under a light at the end of the course. What if you will only have a limited time at the course before the competition, and you don't know the lighting conditions in advance? You might need to calibrate your Boe-Bot on site. A program that makes the piezospeaker beep repeatedly when the Boe-Bot detects bright light and stay quiet when it detects ambient light could be useful for this task. Write and test the program that works with the circuit in Figure 6-4 on page 173.
2. Develop an application that makes the Boe-Bot roam and search for darkness instead of light. This application should utilize the charge transfer circuits in Figure 6-9 on page 182.
3. Develop an application that makes the Boe-Bot roam toward a bright incandescent desk lamp in a room where the only other light sources are fluorescent ceiling lights. The Boe-Bot should be able to roam toward the desk lamp and play a tone when it's under it. This application should utilize the charge transfer circuits in Figure 6-9 on page 182.

Solutions

Q1. The amount of current it allows to pass into its collector and out through it's base.
Q2. The phototransistor's collector and emitter terminals are connected to pins.
Q3. The pin that's closer to the flat spot is the emitter. The pin that's further away from the flat spot is the collector.
Q4. The wavelength of red is closer to the wavelength of infrared, so it should be more sensitive to red.
Q5. V_{P6} increases with more light.
Q6. It supplies the resistor with more or less current.
Q7. Change the 2 kΩ resistor to a higher value.
Q8. 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.
Q9. The voltage decreases.

E1. $V = I \times R = 0.001\ A \times 2000\ \Omega = 2\ V$.

E2. V = I × R → I = V ÷ R = 4.5 ÷ 2000 = 0.00225 A = 2.25 mA.

E3. The BASIC Stamp's threshold voltage is 1.4 V, but the light threshold occurs at 2.8 V. So, the phototransistor delivers a certain current that results in a 2.8 V measurement, in terms of V = I × R, that's 2.8 V = I × 2000 Ω. We need to figure out the resistance to make the voltage 1.4 V for that same current, that's 1.4 V = I × R. To figure out R, rearrange the first equation to determine I; that's I = 2.8 V ÷ 2000 Ω. Then, substitute 2.8 V ÷ 2000 Ω for I in the second equation and solve for R. That's 1.4 V = I × R → 1.4 V = (2.8 V ÷ 2000 Ω) × R → R = 1.4 V ÷ (2.8 V ÷ 2000 Ω) = 2000 Ω × (1.4 V ÷ 2.8 V) = 1000 Ω = 1 kΩ.

E4. 105 → 10 with 5 zeros appended and multiplied by 1 pF. 1,000,000 × 1 pF = (1 × 10^6) × (1 × 10^{-12}) F = 1 × 10^{-6} F = 1 µF.

E5. It would be **`RCTIME 7, 1, tDecay`**

E6. 1.625 × 256 ÷ 5 = 83.2, take 83. Answer: **`PWM 6, 83, 1`**

E7. **`ndShade`** = 500 – (1000 × **`tLeft`** ÷ (**`tLeft`** + **`tRight`**)) = 500 – (1000 × 1001 ÷ (1001 + 1001)) = 500 – 1000/2 = 500 – 500 = 0.

E8. It would be **`DEBUG REP "="\50`**

E9. It would be **`DEBUG CRSRXY 8, 10, "#"`**

P1.

```
' Robotics with the Boe-Bot - CH6P1.bs2
' Chirp periodically if bright light.  Otherwise, stay silent.
' {$STAMP BS2}
' {$PBASIC 2.5}

PAUSE 1000
DEBUG "Program running..."

DO
  IF IN6 = 1 THEN FREQOUT 4, 20, 4000
  PAUSE 100
LOOP
```

P2. The solution for this one is to make a copy of LightSeekingBoeBot.bs2, and add one command to the Main Routine: **`ndShade = -ndShade`**. Add it right after the call to the **`Light_Shade_Info`** subroutine. Then, instead of indicating shade to turn away from, it indicates light to turn away from.

P3. Below is a modified Main Routine from LightSeekingBoeBot.bs2 that roams toward the light and stops when it gets under an incandescent lamp. The key to this one is very simple because LightSeekingBoeBot.bs2 has a **`light`** variable that reaches higher values under bright light. With each repetition of the **`DO…LOOP`**, the **`IF…THEN`** statement checks for values above 320.

For lower light areas and weaker flashlights, you may need to adjust **`IF light > 320 THEN...`** so that it compares the **`light`** variable to a lower value, for example: **`IF light > 250 THEN`**… Decreasing the value the **`IF…THEN`** statement compares to the **`light`** variable to makes it more sensitive; increasing it makes it less sensitive. The value 324 is the highest possible value so don't increase your comparison value above 323.

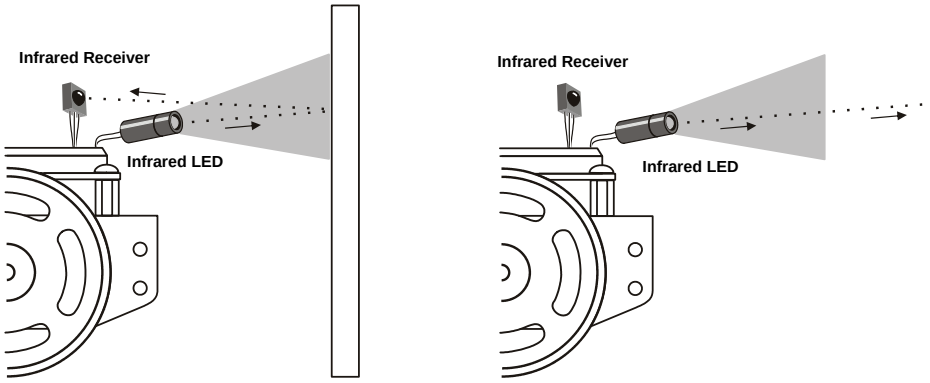
TIP: Use LightSensorValues.bs2 to test and find a value that's between ambient light and the flashlight beam.

```
DO
  GOSUB Light_Shade_Info

  IF light > 320 THEN
    FREQOUT 4, 500, 3000
    PAUSE 500
    FREQOUT 4, 500, 3500
    PAUSE 500
    END
  ENDIF

  IF (ndShade + 500) > 500 THEN
    pulseLeft = 900 - ndShade MIN 650 MAX 850
    pulseRight = 650
  ELSE
    pulseLeft = 850
    pulseRight= 600 - ndShade MIN 650 MAX 850
  ENDIF

  PULSOUT 13, pulseLeft
  PULSOUT 12, pulseRight
LOOP
```

Chapter 7: Navigating with Infrared Headlights

The Boe-Bot can already use whiskers to get around objects it detects when it bumps into them, but wouldn't it be better if the Boe-Bot could just "see" objects and then decide what to do about them? Well, that's exactly what the Boe-Bot can do with infrared headlights and eyes like the ones in Figure 7-1. The infrared headlight is an infrared LED inside a light shield that directs its light forward just like a flashlight. The infrared eye is an infrared receiver that sends the BASIC Stamp high/low signals to indicate whether it detects the infrared LED's light reflected off an object.

Figure 7-1: Infrared Object Detection

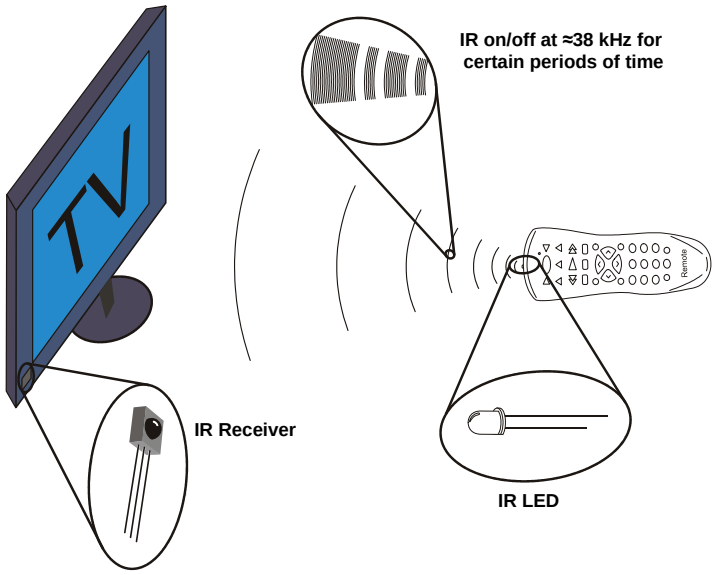


Left: Infrared reflected, obstacle detected. Right: Infrared not reflected, no obstacle detected.

INFRARED LIGHT

Infrared is abbreviated IR and you can think about it as a form of light the human eye cannot detect. For a refresher, take a look at Figure 6-2 on page 171. Devices like the IR LED introduced in this chapter emit infrared light, and devices like the phototransistor from the previous chapter and the infrared receiver from this chapter detect infrared light. Figure 7-2 shows how the infrared LED the Boe-Bot uses as a tiny flashlight is actually the same one you can find in just about any TV remote. The TV remote sends IR messages to your TV, and the microcontroller in your TV picks up those messages with an infrared receiver like the one your Boe-Bot will use to detect IR reflected off of objects.

Figure 7-2: IR LED and Receiver in Your Home



Note that the TV remote sends messages describing which button you press by rapidly flashing its IR LED on/off at a rate in the 38 kHz neighborhood. That's around 38,000 times per second. The IR receiver only responds to infrared if it's flashing on/off 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 the IR receiver can detect, your BASIC Stamp will have to send signals in the 38 kHz range too. The PBASIC language makes short work of that task, with just one line of code to transmit the signal, and a second line to check the IR receiver.

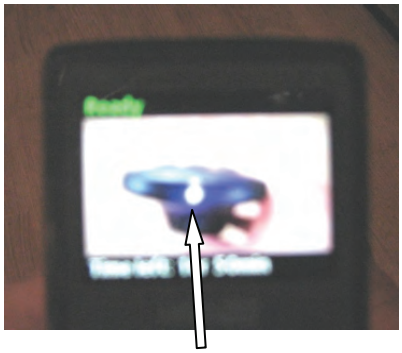
Some fluorescent lights do generate signals that can be detected by the IR receivers. These lights can cause problems for your Boe-Bot's infrared headlights. One of the things you will do in this chapter is develop an infrared interference "sniffer" that you can use to test the fluorescent lights near your Boe-Bot courses.

The light color sensors inside most digital cameras, including cell phones and webcams, can all detect infrared light, which gives us a way to "see" it even if the eye cannot detect it. Figure 7-3 shows an example with a digital camera and a TV remote. When you press

and hold a button on the remote and point its IR LED into the digital camera's lens, it displays the infrared LED as a flashing, bright white light.

Figure 7-3: IR LED in a TV Remote Viewed through a Digital Camera

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 flashing, 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 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.

IR wavelengths and their uses: The wavelength our IR LED transmits is 980 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: BUILDING AND TESTING THE IR OBJECT DETECTORS

In this activity, you will build and test infrared object detectors for the Boe-Bot robot.

- ✓ Gather the parts in the Parts List using Figure 7-4 to help identify the infrared receivers, LEDs, and shield assembly parts.

Parts List:

(2) IR receivers
(2) IR LEDs (clear case)
(2) IR LED shield assemblies
(2) Resistors, 220 Ω (red-red-brown)
(2) Resistors, 1 kΩ (brown-black-red)

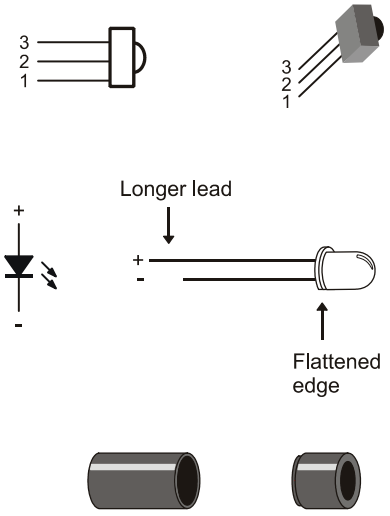


Figure 7-4
New Parts Used in this Chapter

IR receiver (top)

IR LED (middle)

IR LED shield assembly (bottom)

✓ Check Figure 7-5 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 Figure 7-5.

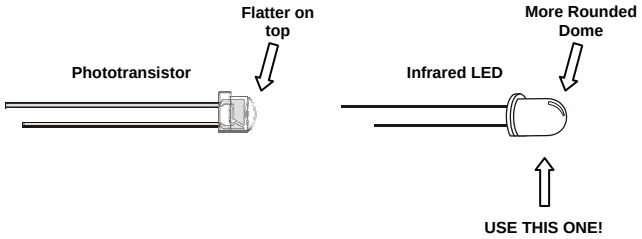


Figure 7-5
Distinguishing Phototransistors from Infrared LEDs

Make sure you have two infrared LEDs.

Building the IR Headlights

✓ Insert the infrared LED into the LED standoff (the larger of the two shield assembly pieces) as shown in Figure 7-6.
✓ Make sure the IR LED snaps into the LED standoff.
✓ Slip the LED shield (the smaller half of the LED shield assembly) over the IR LED's clear plastic case. The ring on one end of the LED shield should fit right into the LED standoff.
✓ Use a small piece of clear tape to make sure the two halves of the shield assembly don't separate during use.

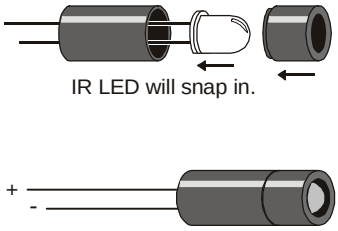


Figure 7-6
Snapping the IR LED into the Shield Assembly

IR Object Detection Circuit

Figure 7-7 shows the IR object detection circuit schematic and Figure 7-8 shows a wiring diagram of the circuit. In the wiring diagram, one IR object detector (IR LED and receiver) is mounted on each corner of the breadboard closest to the very front of the Boe-Bot.

- ✓ Disconnect power from your board and servos.
- ✓ Build the circuit in the Figure 7-7 schematic using the Figure 7-8 wiring diagram as a reference for parts placement.

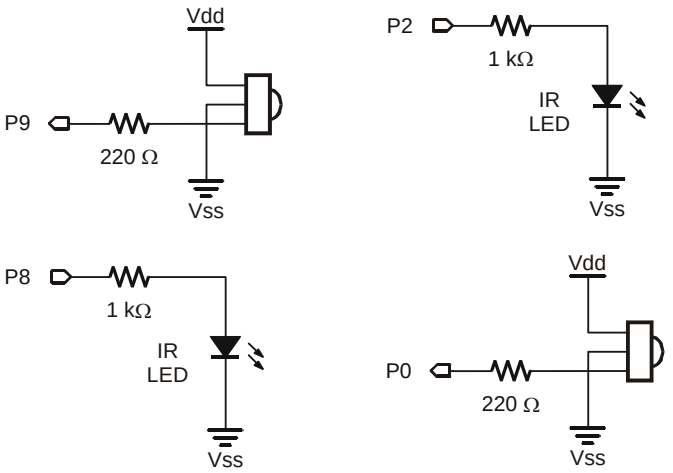


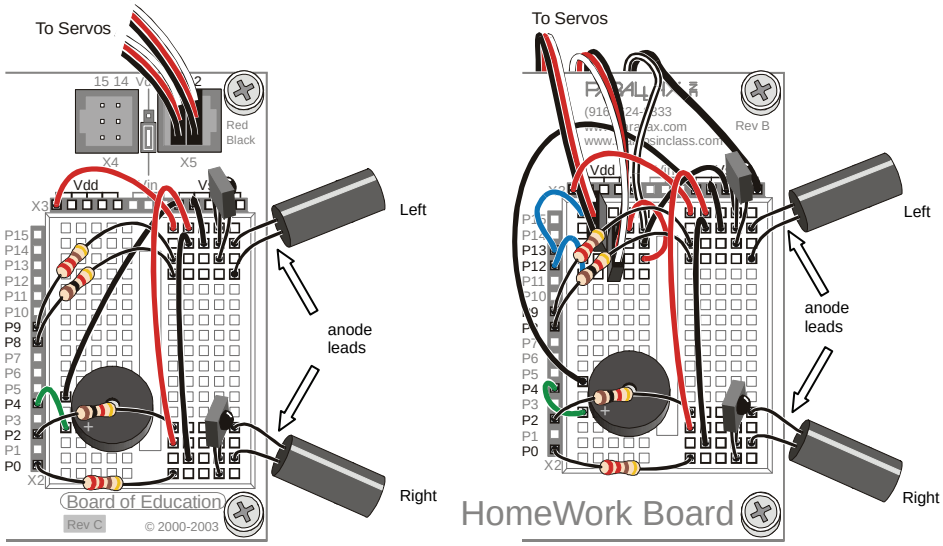
Figure 7-7
Left and Right IR Object Detectors

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. The same conventions applied to the red LEDs in Chapter 2.

In Figure 7-8, the anode lead of each IR LED connects to a 1 kΩ resistor. The cathode lead plugs into the same breadboard row as an IR detector's center pin, and that row is connected to Vss with a jumper wire.

Figure 7-8: Wiring Diagrams for Infrared Emitter and Receiver Circuits



Object Detection Test Code

Your Boe-Bot's infrared receivers are designed to detect infrared light with a 980 nm wavelength that's either flashing on/off or varying in brightness at a rate in the 38 kHz neighborhood. The infrared LED emits 980 nm IR, so that's taken care of. All we need is to make the LED's brightness vary, brighter and then dimmer, at a rate of about 38 kHz. We can do this with the same command we've been using to make the Boe-Bot's speaker play a tone at the beginning of each program—the **FREQOUT** command.

It takes two lines of code to test for the presence or absence of an object using an IR object detection circuit. Here is an example that tests to find out if an object is in front of the Boe-Bot robot's left IR detection circuit.

```
FREQOUT 8, 1, 38500
irDetectLeft = IN9
```

The command **`FREQOUT 8, 1, 38500`** makes the IR LED's brightness vary, getting brighter and dimmer 38500 times per second. It does this for 1 ms; then, **`irDetectLeft = IN9`** stores the IR receiver's output in a variable. The detector's output will be high if it does not detect 38.5 kHz IR reflected off an object, or low if it does. So the value of **`IN9`** that gets copied to the **`irDetectLeft`** variable will be 1 if no object is detected, or 0 if an object is detected.

Always use `irDetectLeft = IN9` right after `FREQOUT 8, 1, 38500`.

The BASIC Stamp only has a brief time window to copy the binary signal it gets from the IR receiver to a variable. The IR receiver sends a low signal while it detects 38.5 kHz IR reflected off an object, which causes **`IN9`** to store 0. When the BASIC Stamp finishes transmitting its **`FREQOUT`** signal and moves on to the next command, it stops sending that 38.5 kHz signal. So the program has to use **`irDetectLeft = IN9`** to catch that zero value before the IR receiver realizes the 38.5 kHz signal stopped. It only takes a fraction of a millisecond for the IR receiver to realize the signal stopped, and after that, its output rebounds to high, and **`IN9`** stores 1 again.

Example Program: TestLeftIr.bs2

- ✓ Reconnect power to your board.
- ✓ Enter, save, and run TestLeftIr.bs2.

```
' Robotics with the Boe-Bot - TestLeftIr.bs2
' Test IR object detection circuits, IR LED to P8 and detector to P9.

' {$STAMP BS2}
' {$PBASIC 2.5}

irDetectLeft   VAR     Bit

DO
  FREQOUT 8, 1, 38500
  irDetectLeft = IN9

  DEBUG HOME, "irDetectLeft = ", BIN1 irDetectLeft
  PAUSE 100
LOOP
```

- ✓ Leave the Boe-Bot connected to its programming cable, because you will be using the Debug Terminal to test your IR object detector.
- ✓ Place an object, such as your hand or a sheet of paper, about an inch from the left IR object detector, in the manner shown in Figure 7-1 on page 221.
- ✓ Verify that the Debug Terminal displays a 0 when you place an object a few inches in front of the IR object detector.
- ✓ Verify that it displays 1 when you remove the object.
- ✓ If the Debug Terminal displays the expected values for object not detected (1) and object detected (0), move on to the Your Turn section.
- ✓ If the Debug Terminal does not display the expected values, try the steps in the Troubleshooting section.

Troubleshooting

If the Debug Terminal does not display the expected values, try this checklist:

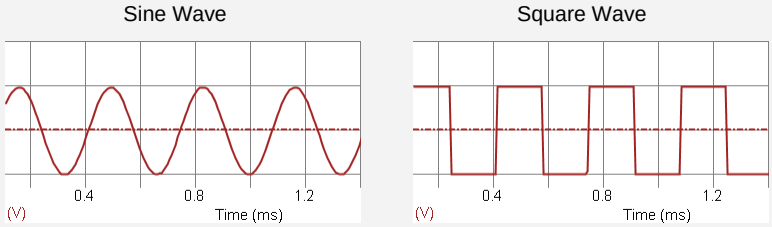
- ✓ Check for circuit and program entry errors. One common error is to use a 10 kΩ resistor (brown-black-orange) instead of 1 kΩ (brown-black-red).
- ✓ Keep the Boe-Bot out of direct sunlight.
- ✓ If you are always getting 0, even when an object is not placed in front of the Boe-Bot, there may be a nearby object that is reflecting the infrared. The surface of the table in front of the Boe-Bot is a common culprit. Move the Boe-Bot so that the IR LED and detector cannot possibly be reflecting off any nearby object.
- ✓ If the reading is 1 most of the time when there is no object in front of the Boe-Bot, but flickers to 0 occasionally, it may mean you have interference from a nearby fluorescent light. Turn off any nearby fluorescent lights and repeat your tests. Also try closing the blinds if you are near a window.
- ✓ If the reading is 1 all of the time, even when an object is placed in front of the Boe-Bot: Although it's not a common mistake, manufacturers occasionally make a batch of LEDs with the longer and shorter leads reversed. If you have already double-checked your wiring and program, try disconnecting the IR LED and reversing its polarity, so that the shorter lead is connected to the 1 kΩ resistor and the longer lead is connected to Vss.
- ✓ One final test you can try is to connect your IR LED circuit to a different I/O pin and adjust your program accordingly. Start with the correct anode/cathode orientation, and if it doesn't work, try reversing it again.

Your Turn – Test the Right IR Object Detector

- ✓ Save TestLeftIr.bs2 as TestRightIr.bs2.
- ✓ Change the **`DEBUG`** command, program title and comments to refer to the right IR object detector.
- ✓ Change the variable name from **`irDetectLeft`** to **`irDetectRight`**. You will need to do this in four places in the program.
- ✓ Change the **`FREQOUT`** command's ***Pin*** argument from 8 to 2.
- ✓ Change the input register monitored by the **`irDetectRight`** variable from **`IN9`** to **`IN0`**.
- ✓ Repeat the testing steps in this activity for the Boe-Bot's right IR object detector.

Sine Waves Synthesized by FREQOUT

The **`FREQOUT`** command transmits a rapid sequence of on/off signals that digitally synthesize voltages to create a sine wave pattern. Sine waves sound much more natural that square waves when played by a speaker. Square waves make more of a buzzing noise.



A **`FREQOUT`** signal contains two sine wave components with two different frequencies. One component's frequency is **`Freq1`**. The second component's frequency 65536 – **`Freq1`**.

When the **`FREQOUT`** command is used to play audible tones, the signal's second frequency is always well above 20 kHz, which is typically the highest pitch that the human ear can detect.

Example 1: **`FREQOUT 4, 2000, 3000`** plays a 3 kHz sine wave tone on the piezospeaker because **`Freq1`** is 3000. The signal contains a second component with a frequency of 65536 – 3000 = 62536 Hz, but the human ear cannot detect it. Since 65536 – 62536 = 3000, you could play the same tone with **`FREQOUT 4, 2000, 62536`**. Although **`Freq1`** is now well outside the human ear's range, the second signal is 3 kHz, so you'll get the same tone out of your piezospeaker.

Example 2: **`FREQOUT 8, 1, 38500`** makes the IR LED's brightness vary at a rate of 38500 Hz so that the IR receiver can detect it. The signal it creates also contains a second sine wave with a frequency of 65536 – 38500 = 27036 Hz, but that signal has no effect on the IR receiver.

ACTIVITY #2: FIELD TESTING FOR OBJECT DETECTION AND INFRARED INTERFERENCE

In this activity, you will build and test indicator LEDs that will tell you if an object is detected without the help of the Debug Terminal. This is handy if you are not near a PC or laptop, and you need to trouble-shoot your IR detector circuits. You will also write a program 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, 38.5 kHz, which in turn causes the lamp to emit a signal at this frequency. When you integrate IR object detection with navigation, this interference can cause some bizarre Boe-Bot behavior!

Rebuilding the LED Indicator Circuits

These are the same LED indicator circuits that you used with the whiskers.

Parts List:

(2) Red LEDs
(2) Resistors, 220 Ω (red-red-brown)

- ✓ Disconnect power from your board and servos.
- ✓ Build the circuit shown in Figure 7-9 using Figure 7-10 as a reference.

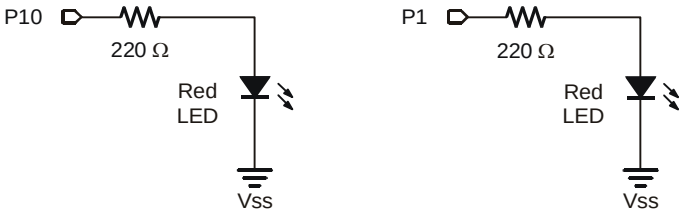
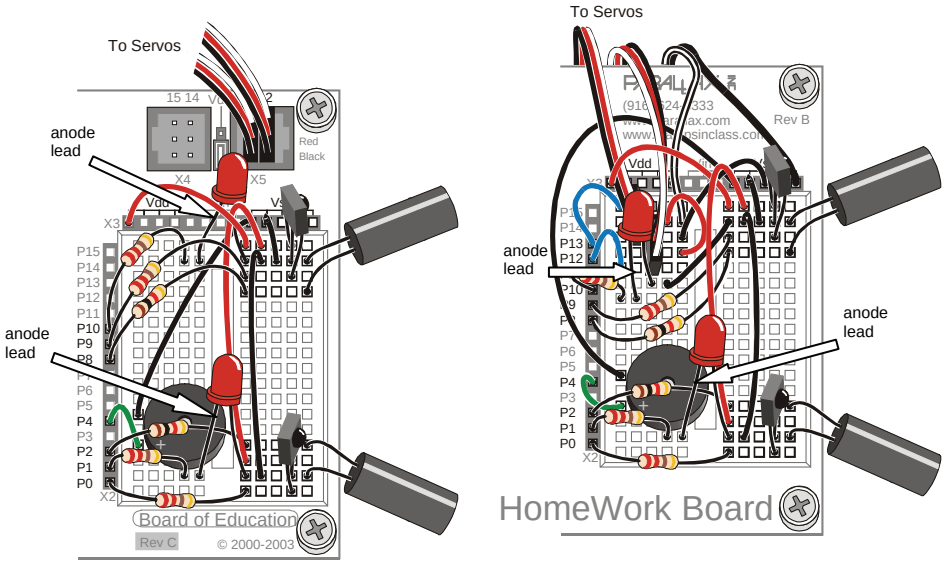


Figure 7-9
Left and Right Indicator LEDs

Figure 7-10: Wiring Diagrams for Infrared Emitter and Receiver Circuits



Board of Education (left) and HomeWork Board (right)

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 program that shows you what the infrared detectors are sensing. You can use this program to verify that all the circuits are working before unplugging the Boe-Bot from its programming cable and testing other objects.

Example Program – TestBothIrAndIndicators.bs2

- ✓ Reconnect power to your board.
- ✓ Enter, save, and run TestBothIrAndIndicators.bs2.
- ✓ Verify that the speaker makes a clear, audible tone while the Debug Terminal displays "Testing piezospeaker…"
- ✓ Use the Debug Terminal to verify that the BASIC Stamp still receives a zero from each IR detector when an object is placed in front of it.

- ✓ Verify that the LED next to each detector emits light when the detector detects an object. If one or both of the LEDs appear not to work, check your wiring and your program.

```
' Robotics with the Boe-Bot - TestBothIrAndIndicators.bs2
' Test IR object detection circuits.

' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

' -----[ Variables ]-------------------------------------------------------

irDetectLeft   VAR     Bit
irDetectRight  VAR     Bit

' -----[ Initialization ]--------------------------------------------------

DEBUG "Testing piezospeaker..."
FREQOUT 4, 2000, 3000

DEBUG CLS,
      "IR DETECTORS", CR,
      "Left   Right", CR,
      "-----  -----"

' -----[ Main Routine ]----------------------------------------------------

DO
  FREQOUT 8, 1, 38500
  irDetectLeft = IN9

  FREQOUT 2, 1, 38500
  irDetectRight = IN0

  IF (irDetectLeft = 0) THEN
    HIGH 10
  ELSE
    LOW 10
  ENDIF

  IF (irDetectRight = 0) THEN
    HIGH 1
  ELSE
    LOW 1
  ENDIF

  DEBUG CRSRXY, 2, 3, BIN1 irDetectLeft,
        CRSRXY, 9, 3, BIN1 irDetectRight

  PAUSE 100
LOOP
```

Your Turn – Remote Testing and Range Testing

You can now use your LED detectors to take your Boe-Bot and test your IR detectors on objects that might not otherwise be in reach of your computer's programming cable.

- ✓ Unplug your Boe-Bot from the programming cable, and take your Boe-Bot to a variety of objects and test the range of the IR detectors.
- ✓ Try the detection range of different colored objects. What color is detected at the furthest range? What color is detected at the closest range?

Sniffing for IR Interference

If you happened to notice that your Boe-Bot let you know it detected something even though nothing was in range, it may mean that a nearby light is generating some IR light at a frequency close to 38.5 kHz. If you try to have a Boe-Bot contest or demonstration under one of these lights, your infrared systems might end up performing very poorly. The last thing anybody wants is to have their robot not perform as advertised during a public demonstration, so make sure to check any prospective demo area with this IR interference "sniffer" program beforehand.

The concept behind this program 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-Bot right now. Likewise, the remotes you use to control these devices all use the same kind of IR LED that's on your Boe-Bot to transmit messages to the IR detector in your TV, VCR, CD/DVD player, etc.

Example Program – IrInterferenceSniffer.bs2

- ✓ Enter, save, and run IrInterferenceSniffer.bs2.
- ✓ Test to make sure the Boe-Bot sounds the alarm when it detects IR interference. If you are in a classroom, you can do this with a separate Boe-Bot that's running TestBothIrAndIndicators.bs2. If you don't have a second Boe-Bot, just use a handheld remote for a TV, VCR, CD/DVD player, or projector. Simply point the remote at the Boe-Bot and press a button. If the Boe-Bot responds by sounding the alarm, you know your IR interference sniffer is working.

```
' Robotics with the Boe-Bot – IrInterferenceSniffer.bs2
' Test fluorescent lights, infrared remotes, and other sources
' of 38.5 kHz IR interference.

' {$STAMP BS2}                                ' Stamp directive.
' {$PBASIC 2.5}                               ' PBASIC directive.

counter        VAR     Nib

DEBUG "IR interference not detected, yet...", CR

DO

  IF (IN0 = 0) OR (IN9 = 0) THEN
    DEBUG "IR Interference detected!!!", CR
    FOR counter = 1 TO 5
      HIGH 1
      HIGH 10
      FREQOUT 4, 50, 4000
      LOW 1
      LOW 10
      PAUSE 20
    NEXT
  ENDIF

LOOP
```

Your Turn – Testing for Fluorescent Lights that Interfere

- ✓ Disconnect your Boe-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.

Always use this IrInterferenceSniffer.bs2 to make sure that any area where you are using the Boe-Bot is free of infrared interference.

ACTIVITY #3: INFRARED 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-Bot's infrared LED headlights brighter, you can also increase its detection range. By resisting electric current less, 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, 2 kΩ (red-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 program that sends a high signal to the LED.

Example Program – P1LedHigh.bs2

- ✓ Enter, save and run P1LedHigh.bs2.
- ✓ Run the program and verify that the LED in the circuit connected to P1 emits light.

```
' Robotics with the Boe-Bot - P1LedHigh.bs2
' Set P1 high to test for LED brightness testing with each of
' these resistor values in turn: 220 ohm , 470 ohm, 1 k ohm.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

HIGH 1

STOP
```

The command **`STOP`** is used here rather than **`END`**, since **`END`** would put the BASIC Stamp into low power mode.

Your Turn – Testing LED Brightness

Remember to disconnect power before you make changes to a circuit. Remember also that the same program will run again when you reconnect power, so you can pick up right where you left off with each test.

- ✓ Note how brightly the P1 LED circuit is glowing with the 220 Ω resistor.

- ✓ Replace the 220 Ω resistor connected to P1 and the right LED's cathode with a 470 Ω resistor. Note now how brightly the LED glows.
- ✓ Repeat for a 2 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.

- ✓ Open and run TestBothIrAndIndicators.bs2 (from page 244).
- ✓ Verify that both detectors are working properly.

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, using 1 kΩ resistor, and record your data in Table 7-1.
- ✓ Replace the 1 kΩ resistors that connect P2 and P8 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 2 kΩ resistors, 470 Ω resistors, and 220 Ω resistors.

Table 7-1: 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 configuration (with 1 kΩ resistors in series with each IR LED).

- ✓ Also, before moving on, make sure to test this last change with TestBothIrAndIndicators.bs2 to verify that both IR object detectors are working properly.

ACTIVITY #4: OBJECT DETECTION AND AVOIDANCE

An interesting thing about the 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, RoamingWithWhiskers.bs2 from page 178 is modified so that it works with the IR detectors.

Converting the Whiskers Program for IR Object Detection/Avoidance

This next example program started as RoamingWithWhiskers.bs2. Aside from adjusting the name and description, two bit variables were added to store the states of the IR detectors.


```
irDetectLeft  VAR     Bit
irDetectRight VAR     Bit
```

A routine was also added to read the IR object detectors.

```
FREQOUT 8, 1, 38500
irDetectLeft = IN9

FREQOUT 2, 1, 38500
irDetectRight = IN0
```

The **IF…THEN** statements were modified so that they look at the variables that store the IR object detections instead of the whisker inputs.

```
IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Left
  GOSUB Turn_Left
ELSEIF (irDetectLeft = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Right
ELSEIF (irDetectRight = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Left
ELSE
  GOSUB Forward_Pulse
ENDIF
```

Example Program – RoamingWithIr.bs2

- ✓ Open RoamingWithWhiskers.bs2
- ✓ Modify it so that it matches the program below.
- ✓ Reconnect power to your board and servos.
- ✓ Save and run it.
- ✓ Verify that, aside from the fact that there's no contact required, it behaves like RoamingWithWhiskers.bs2.

```
' -----[ Title ]-----------------------------------------------------------
' Robotics with the Boe-Bot - RoamingWithIr.bs2
' Adapt RoamingWithWhiskers.bs2 for use with IR object detectors.
' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Variables ]-------------------------------------------------------

  irDetectLeft  VAR     Bit
  irDetectRight VAR     Bit
  pulseCount    VAR     Byte

' -----[ Initialization ]---------------------------------------------------

FREQOUT 4, 2000, 3000                       ' Signal program start/reset.

' -----[ Main Routine ]-----------------------------------------------------

DO
  FREQOUT 8, 1, 38500                       ' Store IR detection values in
  irDetectLeft = IN9                        ' bit variables.

  FREQOUT 2, 1, 38500
  irDetectRight = IN0

  IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
    GOSUB Back_Up                           ' Both detect obstacle
    GOSUB Turn_Left                         ' Back up & U-turn (left twice)
    GOSUB Turn_Left
  ELSEIF (irDetectLeft = 0) THEN            ' Left detects
    GOSUB Back_Up                           ' Back up & turn right
    GOSUB Turn_Right
  ELSEIF (irDetectRight = 0) THEN           ' Right detects
    GOSUB Back_Up                           ' Back up & turn left
    GOSUB Turn_Left
  ELSE                                      ' None detect
    GOSUB Forward_Pulse                     ' Apply a forward pulse
  ENDIF                                     ' and check again
LOOP
```

```
' -----[ Subroutines ]-------------------------------------------------------

Forward_Pulse:                                  ' Send a single forward pulse.
  PULSOUT 13,850
  PULSOUT 12,650
  PAUSE 20
  RETURN

Turn_Left:                                      ' Left turn, about 90-degrees.
  FOR pulseCount = 0 TO 20
    PULSOUT 13, 650
    PULSOUT 12, 650
    PAUSE 20
  NEXT
  RETURN

Turn_Right:
  FOR pulseCount = 0 TO 20                      ' Right turn, about 90-degrees.
    PULSOUT 13, 850
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN

Back_Up:                                        ' Back up.
  FOR pulseCount = 0 TO 40
    PULSOUT 13, 650
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN
```


Your Turn

- ✓ Modify RoamingWithIr.bs2 so that the IR object detectors are checked in a subroutine.

ACTIVITY #5: HIGH-PERFORMANCE IR NAVIGATION

The style of pre-programmed maneuvers that were used in the previous activity were fine for whiskers, but are unnecessarily slow when using the IR LEDs and detectors. You can greatly improve the Boe-Bot's roaming performance by checking for obstacles before delivering each set of pulses to the servos. The program can use the sensor inputs to select the best maneuver for each moment of navigation. That way, the Boe-Bot never turns further than it has to, and it can neatly find its way around obstacles and successfully navigate more complex courses.

Sampling Between Every Pulse to Avoid Collisions

The great thing about detecting an obstacle before bumping into it is that it gives the Boe-Bot some room to navigate around it. The Boe-Bot can apply a pulse to turn away from an object, check again and if the object is still there, apply another pulse to avoid it. The Boe-Bot can keep applying pulses and checking, until it steers clear of the obstacle. Then, it can resume forward pulses. After experimenting with this next example program, you'll likely agree that it's a much better way for the Boe-Bot to roam.

Example Program – FastIrRoaming.bs2

- ✓ Enter, save, and run FastIrRoaming.bs2.

```
' Robotics with the Boe-Bot - FastIrRoaming.bs2
' Higher performance IR object detection assisted navigation

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

irDetectLeft   VAR     Bit                  ' Variable Declarations
irDetectRight  VAR     Bit
pulseLeft      VAR     Word
pulseRight     VAR     Word

FREQOUT 4, 2000, 3000                       ' Signal program start/reset.

DO                                          ' Main Routine

  FREQOUT 8, 1, 38500                       ' Check IR Detectors
  irDetectLeft = IN9
  FREQOUT 2, 1, 38500
  irDetectRight = IN0
                                            ' Decide how to navigate.
  IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
    pulseLeft = 650
    pulseRight = 850
  ELSEIF (irDetectLeft = 0) THEN
    pulseLeft = 850
    pulseRight = 850
  ELSEIF (irDetectRight = 0) THEN
    pulseLeft = 650
    pulseRight = 650
  ELSE
    pulseLeft = 850
    pulseRight = 650
  ENDIF
```

```
  PULSOUT 13,pulseLeft                          ' Apply the pulse.
  PULSOUT 12,pulseRight
  PAUSE 15

LOOP                                            ' Repeat Main Routine
```

How FastIrRoaming.bs2 Works

This program takes a slightly different approach to applying pulses. Aside from the two bits used to store the IR detector outputs, it uses two word variables to set the pulse durations delivered by the **PULSOUT** command.

```
irDetectLeft   VAR     Bit
irDetectRight  VAR     Bit
pulseLeft      VAR     Word
pulseRight     VAR     Word
```


Inside the **DO…LOOP**, the **FREQOUT** commands are used to send a 38.5 kHz IR signal to each IR LED. Immediately after the 1 ms burst of IR is sent, a bit variable stores the output state of the IR detector. This is necessary, because if you wait any longer than a command's worth of time, the IR detector will return to the not detected (1 state), regardless of whether or not it detected an object.

```
FREQOUT 8, 1, 38500
irDetectLeft = IN9
FREQOUT 2, 1, 38500
irDetectRight = IN0
```

In the **IF…THEN** statements, instead of delivering pulses or calling navigation routines, this program sets variable values that will be used in **PULSOUT** commands' ***Duration*** arguments.

```
IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
  pulseLeft = 650
  pulseRight = 850
ELSEIF (irDetectLeft = 0) THEN
  pulseLeft = 850
  pulseRight = 850
ELSEIF (irDetectRight = 0) THEN
  pulseLeft = 650
  pulseRight = 650
ELSE
  pulseLeft = 850
  pulseRight = 650
ENDIF
```

Before the **DO…LOOP** repeats, the last thing to do is to deliver pulses to the servos. Notice that the **PAUSE** command is no longer 20. Instead, it's 15 since roughly 5 ms is taken checking the IR LEDs.

```
PULSOUT 13,pulseLeft                    ' Apply the pulse.
PULSOUT 12,pulseRight
PAUSE 15
```

Your Turn

- ✓ Save FastIrRoaming.bs2 as FastIrRoamingYourTurn.bs2.
- ✓ Use the LEDs to broadcast that the Boe-Bot has detected an object.
- ✓ Try modifying the values that pulseLeft and pulseRight are set to so that the Boe-Bot does everything at half speed.

ACTIVITY #6: THE DROP-OFF DETECTOR

Up until now, the Boe-Bot has mainly been programmed to take evasive maneuvers when an object is detected. There are also applications where the Boe-Bot must take evasive action when an object is not detected. For example, if the Boe-Bot is roaming on a table, its IR detectors might be looking down at the table surface as shown in Figure 7-11. The program should make it continue forward so long as both IR detectors can "see" the surface of the table.

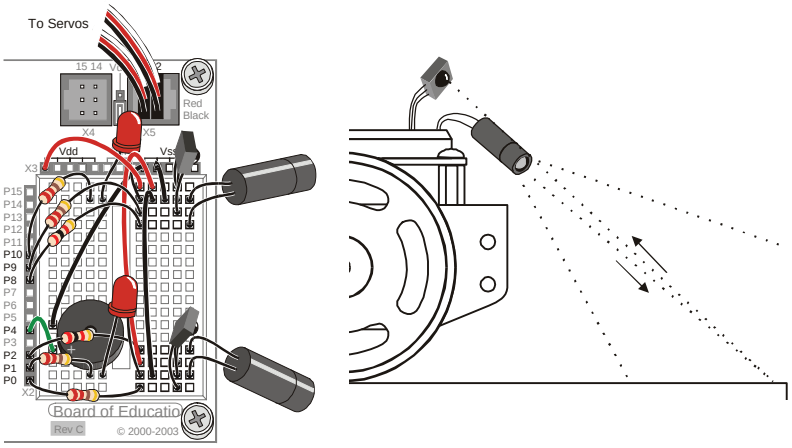


Figure 7-11
IR Object detectors Directed Downwards to Scan for a Drop-Off

Top view (left); side view (right).

- ✓ Disconnect power from your board and servos.
- ✓ Point your IR object detectors downward and outward as shown in Figure 7-11.

Recommended Materials:

(1) Roll of black vinyl electrical tape, ¾″ (19 mm) wide.
(1) Sheet of white poster board, 22 x 28 in (56 x 71 cm).

Simulating a Drop-Off with Electrical Tape

A sheet of white poster board with a border made of electrical tape makes for a handy way to simulate the drop-off presented by a table edge, with much less risk to your Boe-Bot.

- ✓ Build a course similar to the electrical tape delimited course shown in Figure 7-12. Use at least three strips of electrical tape, edge to edge with no paper visible between the strips.
- ✓ Replace your 1 kΩ resistors with 2 kΩ resistors (red-black-red) to connect P2 to its IR LED and P8 to its IR LED. We want the Boe-Bot to be nearsighted for this activity.
- ✓ Reconnect power to your board.
- ✓ Run the program IrInterferenceSniffer.bs2 (page 234) to make sure that nearby fluorescent lighting will not interfere with your Boe-Bot's IR detectors.
- ✓ Use the TestBothIrAndIndicators.bs2 (page 232) to make sure that the Boe-Bot detects the poster board but does not detect the electrical tape.

If the Boe-Bot still "sees" the electrical tape too clearly, here are a few remedies:

- ✓ Try adjusting the IR detectors and LEDs downward at various 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-Bot more nearsighted.
- ✓ Adjust the **FREQOUT** command with different ***Freq1*** arguments. Here are some arguments that will make the Boe-Bot more nearsighted: 38250, 39500, 40500

If you are using older IR LEDs, the Boe-Bot might actually be having problems with being **too nearsighted.** Here are some remedies that will increase the Boe-Bot's sensitivity to objects and make it more far sighted:

- ✓ Try 1 kΩ (brown-black-red) or 470 Ω (yellow-violet-brown) or even 220 Ω (red-red-brown) resistors in series with the IR LEDs instead of 2 kΩ.

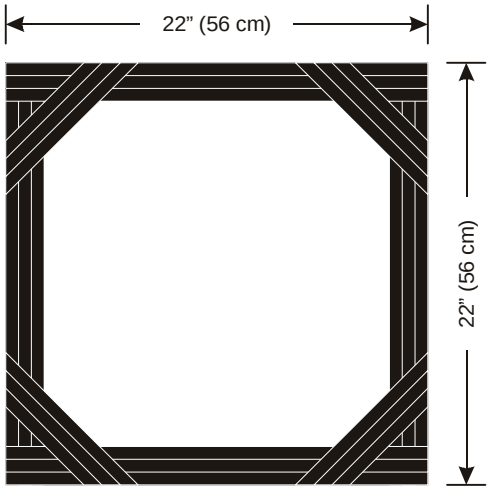


Figure 7-12
Electrical Tape Outline Simulates Tabletop Edge

If you try a tabletop after success with the electrical tape course:

- ✓ Remember to follow the same steps you followed before running the Boe-Bot in the electrical tape delimited course!

Make sure to be the spotter for your Boe-Bot. Be ready as your Boe-Bot roams the tabletop:

- ✓ Always be ready to pick your Boe-Bot up from above as it approaches the edge of the table it's navigating. If the Boe-Bot tries to drive off the edge, pick it up before it takes the plunge. Otherwise, your Boe-Bot might become a Not-Bot!
- ✓ Your Boe-Bot may detect you if you are standing in its line of sight. Its current program 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.

Programming for Drop-Off Detection

For the most part, programming your Boe-Bot to navigate around a table top without going over the edge is a matter of adjusting the **`IF...THEN`** statements from FastIrNavigation.bs2. The main adjustment is that the servos should be directed to make the Boe-Bot roll forward when **`irDetectLeft`** and **`irDetectRight`** are both 0, indicating that an object (the table's surface) has been detected. The Boe-Bot also has to turn away from a detector that indicates it has not detected an object. For example, if **`irDetectLeft`** is 1, the Boe-Bot had better turn right.

A second feature of a program for turning away from drop-offs is adjustable distance. You may want your Boe-Bot to only take one pulse forward between checking the

detectors, but as soon as a drop-off is detected, you may want your Boe-Bot to take several pulses worth of turn before checking the detectors again.

Just because you are taking multiple pulses in an evasive maneuver, it doesn't mean you have to return to whiskers-style navigation. Instead, you can add a **pulseCount** variable that you can use to set to the number of pulses to deliver for a maneuver. The **PULSOUT** command can be placed inside a **FOR…NEXT** loop that executes **FOR 1 TO pulseCount** pulses. For one pulse forward, **pulseCount** can be 1, for ten pulses left, **pulseCount** can be set to 10, and so on.

Example Program – AvoidTableEdge.bs2

- ✓ Open FastIrNavigation.bs2 and save it as AvoidTableEdge.bs2.
- ✓ Modify the program so that it matches the example program. This will involve adding variables, modifying the **IF…THEN** statements, and nesting the **PULSOUT** commands inside a **FOR…NEXT** loop. Be careful to make sure that all the **pulseLeft** and **pulseRight** variable values inside the **IF…THEN** statement are properly adjusted. Their values are different from the ones in FastIrNavigation.bs2 because the rules of the course are different.
- ✓ Reconnect your board and servos.
- ✓ Test the program on your electrical tape delimited course.
- ✓ If you decide to try a tabletop, remember to follow the testing and spotting tips discussed earlier.


```
' Robotics with the Boe-Bot - AvoidTableEdge.bs2
' IR detects object edge and navigates to avoid drop-off.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

irDetectLeft   VAR     Bit                  ' Variable declarations.
irDetectRight  VAR     Bit
pulseLeft      VAR     Word
pulseRight     VAR     Word
loopCount      VAR     Byte
pulseCount     VAR     Byte

FREQOUT 4, 2000, 3000                       ' Signal program start/reset.
```

```
DO                                              ' Main Routine.

  FREQOUT 8, 1, 38500                           ' Check IR detectors.
  irDetectLeft = IN9
  FREQOUT 2, 1, 38500
  irDetectRight = IN0
                                                ' Decide navigation.

  IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
    pulseCount = 1                              ' Both detected,
    pulseLeft = 850                             ' one pulse forward.
    pulseRight =  650
  ELSEIF (irDetectRight = 1) THEN               ' Right not detected,
    pulseCount = 10                             ' 10 pulses left.
    pulseLeft = 650
    pulseRight =  650
  ELSEIF (irDetectLeft = 1) THEN                ' Left not detected,
    pulseCount = 10                             ' 10 pulses right.
    pulseLeft = 850
    pulseRight =  850
  ELSE                                          ' Neither detected,
    pulseCount = 15                             ' back up and try again.
    pulseLeft = 650
    pulseRight =  850
  ENDIF

  FOR loopCount = 1 TO pulseCount               ' Send pulseCount pulses
    PULSOUT 13,pulseLeft
    PULSOUT 12,pulseRight
    PAUSE 20
  NEXT

LOOP
```

How AvoidTableEdge.bs2 Works

Since this program is a modified version of FastIrRoaming.bs2, only changes to the program are discussed here.

A **`FOR…NEXT`** loop is added to the program to control how many pulses are delivered each time through the main (**`DO…LOOP`**) routine. Two variables are added, **`loopCount`** functions as an index for a **`FOR…NEXT`** loop and **`pulseCount`** is used as the ***EndValue*** argument.

```
loopCount       VAR     Byte
pulseCount      VAR     Byte
```

The **IF…THEN** statements now set the value of **pulseCount** as well as **pulseRight** and **pulseLeft**. If both detectors can see the table, take one cautious pulse forward.

```
IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
    pulseCount = 1
    pulseLeft = 850
    pulseRight =  650
```

Else, if the right IR detector does not see the tabletop, rotate left 10 pulses.

```
ELSEIF (irDetectRight = 1) THEN
  pulseCount = 10
  pulseLeft = 650
  pulseRight =  650
```

Else, if the left IR detector does not see the tabletop, rotate right 10 pulses.

```
ELSEIF (irDetectLeft = 1) THEN
  pulseCount = 10
  pulseLeft = 850
  pulseRight =  850
```

Else, if neither detector can see the table top, back up 15 pulses and try again, hoping that one of the detectors will see the drop-off before the other.

```
ELSE
  pulseCount = 15
  pulseLeft = 650
  pulseRight =  850
ENDIF
```

Now that the value of **pulseCount**, **pulseLeft**, and **pulseRight** are set, this **FOR…NEXT** loop delivers the specified number of pulses for the maneuver determined by the **pulseLeft** and **pulseRight** variables.

```
FOR loopCount = 1 TO pulseCount
  PULSOUT 13,pulseLeft
  PULSOUT 12,pulseRight
  PAUSE 20
NEXT
```

Your Turn

You can experiment with setting different **pulseLeft**, **pulseRight**, and **pulseCount** values inside the **IF…THEN** statement. For example, if the Boe-Bot doesn't turn as far, it

may actually track the edge of the electrical tape delimited course. Pivoting backward instead of rotating in place may also lead to some interesting behaviors.

- ✓ Modify AvoidTableEdge.bs2 so that it follows the edge of the electrical tape delimited course by adjusting the **`pulseCount`** values so that the Boe-Bot doesn't turn too far away from the edge.
- ✓ Experiment with pivoting as a way to make the Boe-Bot roam inside the perimeter instead of following the edge.

SUMMARY

This chapter covered a unique technique for infrared object detection that uses the infrared LED found in common handheld remotes, and the infrared detector found in TVs, CD/DVD players, and other appliances that are controlled by these remotes. By shining infrared into the Boe-Bot's path and looking for its reflection, object detection can be accomplished without physically contacting the object. Infrared LED circuits are used to send a 38.5 kHz signal with the help of a property of the **`FREQOUT`** command called a harmonic, which is inherent to digitally synthesized signals.

An infrared detection indicator program was introduced for remote (not connected to the PC) testing of the IR LED/detector pairs. An infrared interference sniffer program was also introduced to help detect interference that can be generated by some fluorescent light fixtures. Since the signals sent by the IR detectors are so similar to the signals sent by the whiskers, RoamingWithWhiskers.bs2 was adapted to the infrared detectors. A program that checks the IR detectors between each servo pulse was introduced to demonstrate a higher performance way of roaming without colliding into objects. This program was then modified to avoid the edge of an electrical tape delimited area. Since electrical tape absorbs infrared, framing a large sheet of construction paper emulates the drop-off that is seen at a table edge without the danger to the actual Boe-Bot.

Questions

1. What is the frequency of the signal sent by **`FREQOUT 2, 1, 38500`**? What is the value of the second frequency sent by that command? How long are these signals sent for? What I/O pin does the IR LED circuit have to be connected to in order to broadcast this signal?
2. What command has to immediately follow the **`FREQOUT`** command in order to accurately determine whether or not an object has been detected?

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.bs2 so that it only monitors one of the IR detectors.
2. Explain the function of **`pulseCount`** in AvoidTableEdge.bs2.

Projects

1. Design a Boe-Bot application that sits still until you wave your hand in front of it, then it starts roaming.
2. Design a Boe-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-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.

Solutions

Q1. 38.5 kHz is the frequency of the signal. The second frequency = 65536 – 38500 = 27036 Hz. The signals are sent for 1 millisecond, and the IR LED must be connected to I/O Pin 2.

Q2. The command which stores the detector's output in a variable. For example, **`irDetectLeft = IN9`**.

Q3. A low signal means IR at 38.5 kHz was detected, thus, an object was detected. A high signal means no IR at 38.5kHz was detected, so, no object.

Q4. Electrically speaking, for both red and infrared LEDs, a smaller resistor will cause the LED to glow more brightly. A bigger resistor results in dimmer LEDs. In terms of results, brighter IR LEDs make it possible to detect objects that are farther away.

E1. Change the **`IF…THEN`** to read.

```
IF (IN0 = 0) THEN
```

E2. The program sets this variable to 1 when it's taking a forward pulse. That way, as the Boe-Bot moves forward, it checks for a drop-off between each pulse. When it detects a drop-off, it executes a turn for a certain number of pulses, which is also determined by the value of the **`pulseCount`** variable.

P1. The FastIrRoaming.bs2 program can be combined with a **`DO…LOOP UNTIL`** loop that does nothing until it detects an object. A sample solution is shown below.

```
' -----[ Title ]-----------------------------------------------------
' Robotics with the Boe-Bot - MotionActivatedBoeBot.bs2
' Boe-Bot starts roaming when hand is waved in front of IR detectors.

' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Variables ]-------------------------------------------------

irDetectLeft   VAR     Bit                  ' Variable Declarations
irDetectRight  VAR     Bit
pulseLeft      VAR     Word
pulseRight     VAR     Word

' -----[ Initialization ]--------------------------------------------

DEBUG "Program Running!"
FREQOUT 4, 2000, 3000                       ' Signal program
start/reset.

' -----[ Main Routine ]----------------------------------------------

Main:
' Loop until something is detected
  DO
    GOSUB Check_IRs
  LOOP UNTIL (irDetectLeft = 0) OR (irDetectRight = 0)
' Now start roaming -- this code from FastIrRoaming.bs2
  DO
    IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
      pulseLeft = 650                       ' Both detect
      pulseRight = 850                      ' Back up
    ELSEIF (irDetectLeft = 0) THEN          ' Left detect
      pulseLeft = 850                       ' Turn right
      pulseRight = 850
    ELSEIF (irDetectRight = 0) THEN         ' Right detect
      pulseLeft = 650                       ' Turn left
      pulseRight = 650
    ELSE                                    ' Nothing detected
      pulseLeft = 850                       ' Go forward
      pulseRight = 650
```

```
    ENDIF

    PULSOUT 13, pulseLeft                   ' Apply the pulse.
    PULSOUT 12, pulseRight
    PAUSE 15

    GOSUB Check_IRs                         ' Check IRs again
  LOOP

' -----[ Subroutines ] ------------------------------------------------

Check_IRs:
  FREQOUT 8, 1, 38500                       ' Check IR Detectors
  irDetectLeft = IN9
  FREQOUT 2, 1, 38500
  IrDetectRight = IN0
  RETURN
```


P2. This behavior is in many ways the opposite of the roaming behavior covered in this chapter. Instead of avoiding objects, the Boe-Bot tries to go toward the objects. For this reason, the main code can be derived from "FastIrRoaming.bs2", with a bit added that spins the Boe-Bot slowly until an object is detected. In the solution below, once the Boe-Bot has spied an object, it will continue forward even if the detectors both read "no object" (1) for a few loops. This is because, as the Boe-Bot is maneuvering toward the object, sometimes the detectors read "no object" for brief moments, but this is not reason enough to give up the chase.

```
' Robotics with the Boe-Bot - SumoBoeBot.bs2
' Search for object, lock onto it and push it.
' {$STAMP BS2}
' {$PBASIC 2.5}

irDetectLeft   VAR     Bit                  ' Left IR reading
irDetectRight  VAR     Bit                  ' Right IR reading
pulseLeft      VAR     Word                 ' pulse values for servos
pulseRight     VAR     Word

' -----[ Initialization ]----------------------------------------------

DEBUG "Program Running!"
FREQOUT 4, 2000, 3000                       ' Signal start/reset.

' -----[ Main Routine ]------------------------------------------------

Main:

' Spin around slowly until an object is spotted
```

```
  DO
    PULSOUT 13, 790                         ' Rotate slowly
    PULSOUT 12, 790
    PAUSE 15                                ' 5 ms for detectors
    GOSUB Check_IRs                         ' While looking for object
  LOOP UNTIL (irDetectLeft = 0) OR (irDetectRight = 0)

' Now figure out exactly where the object is and go toward it
  DO
  ' Object in both detectors -- go forward
    IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
      pulseLeft =  850                      ' Forward
      pulseRight = 650
  ' Object on left - go left
    ELSEIF (irDetectLeft = 0) THEN
      pulseLeft =  650                      ' Left toward object
      pulseRight = 650
  ' Object on right - go right
    ELSEIF (irDetectRight = 0) THEN
      pulseLeft =  850                      ' Right toward object
      pulseRight = 850
  ' No object -- go forward anyway, because the detectors will
    ELSE                                    ' momentarily show
      pulseLeft =  850                      ' "no object" as the
      pulseRight = 650                      ' Boe-Bot is adjusting
    ENDIF                                   ' its position.

    PULSOUT 13,pulseLeft                    ' Apply the pulse.
    PULSOUT 12,pulseRight
    PAUSE 15                                ' 5 ms for detectors

  ' Check IRs again in case object is moving
    GOSUB Check_IRs
  LOOP

' -----[ Subroutines ] ------------------------------------------------

Check_IRs:
  FREQOUT 8, 1, 38500                       ' Check IR Detectors
  irDetectLeft = IN9
  FREQOUT 2, 1, 38500
  IrDetectRight = IN0
  RETURN
```

P3. The key to solving this problem is to combine "FastIrRoaming.bs2" and "IrInterferenceSniffer.bs2" in a single program.

```
' -----[ Title ]-----------------------------------------------------
' Robotics with the Boe-Bot - RoamAndSniffBoeBot.bs2
' Boe-Bot roams around and sounds alarm when IR detected.
```

```
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Variables ]-------------------------------------------------

irDetectLeft   VAR     Bit                  ' Left IR sensor reading
irDetectRight  VAR     Bit                  ' Right IR sensor reading
pulseLeft      VAR     Word                 ' Pulses sent to servos
pulseRight     VAR     Word
counter        VAR     Nib                  ' Loop counter

' -----[ Initialization ]--------------------------------------------

DEBUG "Program Running!"
FREQOUT 4, 2000, 3000                       ' Signal program
start/reset.

' -----[ Main Routine ]----------------------------------------------

Main:
DO
  GOSUB Roam
  GOSUB Sniff
LOOP

' -----[ Subroutines ] ----------------------------------------------

Sniff:                              ' From IrInterferenceSniffer.bs2
  IF (IN0 = 0) OR (IN9 = 0) THEN
    FOR counter = 1 TO 5                    ' Beep 5 times
      HIGH 1                                ' and flash LEDs
      HIGH 10
      FREQOUT 4, 50, 4000
      LOW 1
      LOW 10
      PAUSE 20
    NEXT
  ENDIF
  RETURN

Roam:                                       ' From FastIrRoaming.bs2
  FREQOUT 8, 1, 38500                       ' Check IR Detectors
  irDetectLeft = IN9
  FREQOUT 2, 1, 38500
  irDetectRight = IN0
                                            ' Decide how to navigate.
  IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
    pulseLeft = 650
    pulseRight = 850
  ELSEIF (irDetectLeft = 0) THEN
    pulseLeft = 850
    pulseRight = 850
```

```
  ELSEIF (irDetectRight = 0) THEN
    pulseLeft = 650
    pulseRight = 650
  ELSE
    pulseLeft = 850
    pulseRight = 650
  ENDIF

  PULSOUT 13,pulseLeft                      ' Apply the pulse.
  PULSOUT 12,pulseRight
  PAUSE 15
  RETURN
```

Chapter 8: Robot Control with Distance Detection

In Chapter 7, we used the infrared LEDs and receivers to detect whether an object is in the Boe-Bot's way without actually touching it. Wouldn't it be nice to also know how far away the object is? This is usually a task for sonar, which sends a pulse of sound out and records how long it takes for the echo to come back. The time it takes for the echo to come back can then be used to calculate how far away the object is. There is, however, a way to accomplish distance detection with the very same circuit you used in the previous chapter. With your Boe-Bot able to determine the distance of an object, it can be programmed to follow a moving object without colliding into it. The Boe-Bot can also be programmed to follow black tracks on a white background.

DETERMINING DISTANCE WITH THE SAME IR LED/DETECTOR CIRCUIT

You will use the same circuit from the previous chapter to detect distance.

- ✓ If the circuit is still built on your Boe-Bot, make sure your IR LED's have 1 kΩ resistors in series.
- ✓ If you already disassembled the circuit from the previous chapter, repeat the steps in Chapter 7, Activity #1, on page 223.

Recommended Equipment and Materials:

(1) Ruler
(1) Sheet of paper

ACTIVITY #1: TESTING THE FREQUENCY SWEEP

Figure 8-1 shows an excerpt from one specific brand of IR detector's datasheet (Panasonic PNA4602M; a different brand may have been used in your kit). This excerpt is a graph that shows how much less sensitive this IR detector becomes if the IR signal it receives flashes on/off at a frequency other than 38.5 kHz. For example, if you send it IR flashed on/off at 40 kHz, it's only 50% as sensitive as it would be at 38.5 kHz. If the IR is flashed on/off at 42 kHz, the detector is only 20% as sensitive. Especially for frequencies that make the detector less sensitive, the object has to be closer to make the reflected IR brighter for the detector to detect it.

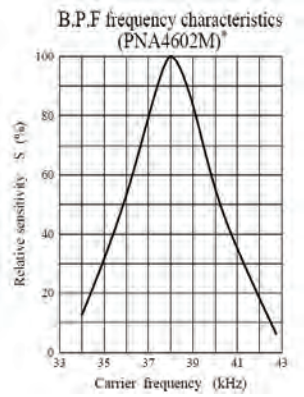


Figure 8-1
Filter Sensitivity Depends on Carrier Frequency

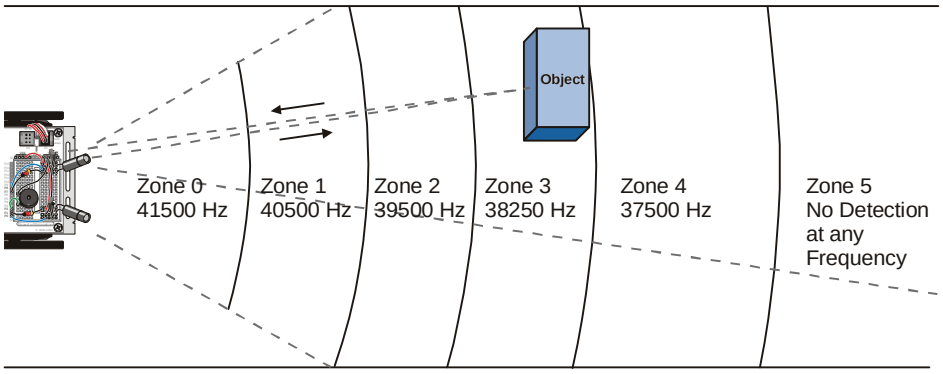
Another way to think about it is that the most sensitive frequency will detect the objects that are the farthest away, while less sensitive frequencies can only be used to detect closer objects. This makes distance detection simple. Pick 5 frequencies, then test them from most sensitive to least sensitive. Try at 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, you can infer the distance.

Frequency Sweep is the technique of testing a circuit's output using a variety of input frequencies.

Programming Frequency Sweep for Distance Detection

Figure 8-2 shows an example of how the Boe-Bot can test for distance using frequency. In this example, the object is in Zone 3. That means that the object can be detected when 37500 and 38250 Hz is transmitted, but it cannot be detected with 39500, 40500, and 41500 Hz. If you were to move the object into Zone 2, then the object can be detected when 37500, 38250, and 39500 Hz are transmitted, but not when 40500 and 41500 Hz are transmitted.

Figure 8-2 Distance Detection Frequencies and Zones for the Boe-Bot



You might be wondering why the value of zone 4 is 37.5 kHz and not 38.5 kHz. The reason they are not the values that you would expect based on the % sensitivity graph is because the **FREQOUT** command transmits a slightly more powerful signal at 37.5 kHz than it does at 38.5 kHz. The frequencies listed in Figure 8-2 are frequencies you will program the BASIC Stamp to use to determine the distance of an object.

In order to test the IR detector at each frequency, you will need to use **FREQOUT** to send five different frequencies and test at each frequency to find out whether the IR detector could see the object. The steps between each frequency are not quite even enough to use the **FOR…NEXT** loop's **STEP** option. You could use **DATA** and **READ**, but that would be cumbersome. You could use five different **FREQOUT** commands, but that would be a waste of code space. Instead, the best approach for storing a short list of values that you want to use in sequence is a command called **LOOKUP**. The syntax for the **LOOKUP** command is:

LOOKUP *Index*, [*Value0*, *Value1*, …*ValueN*], *Variable*

If the ***Index*** argument is 0, ***Value0*** from the list inside the square braces will be placed in ***Variable***. If ***Index*** is 1, ***Value1*** from the list will be placed in ***Variable***. There could be up to 256 values in the list, but for the next example program, we will only need 5. Here is how it will be used:

```
FOR freqSelect = 0 TO 4
  LOOKUP freqSelect,[37500,38250,39500,40500,41500],irFrequency
  FREQOUT 8,1, irFrequency
  irDetect = IN9
    ' Commands not shown...
NEXT
```

The first time through the **FOR…NEXT** loop, **freqSelect** is 0, so the **LOOKUP** command places the value 37500 in the **irFrequency** variable. Since **irFrequency** contains 37500 after the **LOOKUP** command, the **FREQOUT** command sends that frequency to the IR LED connected to P8. As in the previous chapter, the value of **IN9** is then saved in the **irDetect** variable. The second time through the **FOR…NEXT** loop, the value of **freqSelect** is now 1, which means the **LOOKUP** command places 38250 into the **irFrequency** variable, and the process is repeated for this higher frequency. The third time through, it's repeated again with 39500, and so on. The result is remarkable, especially considering you are using parts that were designed for a completely different purpose, to make IR communication between a handheld remote and a television possible.

Example Program – TestLeftFrequencySweep.bs2

TestLeftFrequencySweep.bs2 does two things. First, it tests the left IR object detector (connected to P8 and P9) to make sure it is functioning properly for distance detection. However, it also demonstrates how the frequency sweep illustrated in Figure 8-2 is accomplished.

When you run the program, the Debug Terminal will display your zone measurement. There are many possible yes-no patterns that can be generated; two are shown in Figure 8-3. The test patterns will vary depending on the characteristics of the filter inside the IR detector.

The program determines which zone the detected object is in by counting the number of "No" occurrences. Notice that even though the two Debug Terminal test patterns in Figure 8-3 are different, they both have three "Yes" and two "No" occurrences. Therefore, "Zone 2" is the location of the object detected in both examples.

- ✓ Enter, save, and run TestLeftFrequencySweep.bs2.
- ✓ Use a sheet of paper or card facing the IR LED/detector to test the distance detection.
- ✓ Start with the sheet very close to the IR LED, perhaps ¼ in (or 1 cm) away from the IR LED. Your Zone in the Debug Terminal should either be 0 or 1.
- ✓ Gradually move the sheet of paper away from the IR LED and make a note of each distance that causes the zone value to get larger.

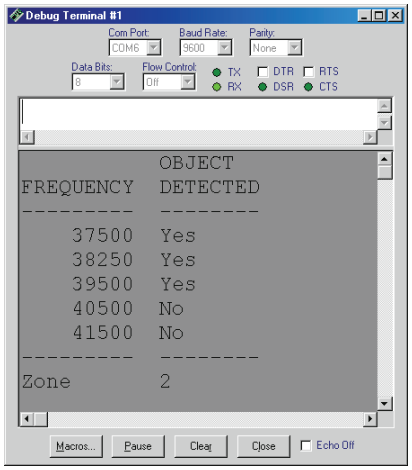
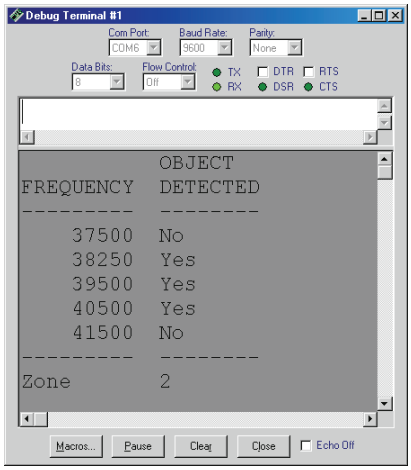


Figure 8-3
Testing Distance Detection Output Examples



Keep in mind that these distance measurements are relative and not necessarily precise or evenly spaced. However, they will give the Boe-Bot a good enough sense of object distance for following, tracking, and other activities.

Zones 1-4 typically fall in the range of 6 to 12 in (15 to 30 cm) for the shielded LEDs with a 1 kΩ resistor. As long as objects can be detected up to 4 in (10 cm) away, the experiments in this chapter will work. If the distance detector range is less than that, try reducing your series resistance from 1 kΩ to 470 Ω or 220 Ω.

```
' -----[ Title ]-----------------------------------------------------------------
' Robotics with the Boe-Bot - TestLeftFrequencySweep.bs2
' Test IR detector distance responses to frequency sweep.

' {$STAMP BS2}                                 ' Stamp directive.
' {$PBASIC 2.5}                                ' PBASIC directive.

' -----[ Variables ]-------------------------------------------------------------

freqSelect     VAR     Nib
irFrequency    VAR     Word
irDetect       VAR     Bit
distance       VAR     Nib

' -----[ Initialization ]--------------------------------------------------------

DEBUG CLS,
      "           OBJECT", CR,
      "FREQUENCY  DETECTED", CR,
      "---------  --------"
```

```
' -----[ Main Routine ]----------------------------------------------------

DO

  distance = 0

  FOR freqSelect = 0 TO 4

    LOOKUP freqSelect,[37500,38250,39500,40500,41500], irFrequency
    FREQOUT 8,1, irFrequency
    irDetect = IN9
    distance = distance + irDetect

    DEBUG CRSRXY, 4, (freqSelect + 3), DEC5 irFrequency
    DEBUG CRSRXY, 11, freqSelect + 3

    IF (irDetect = 0) THEN DEBUG "Yes" ELSE DEBUG "No "

    PAUSE 100

  NEXT

  DEBUG CR,
        "---------  --------", CR,
        "Zone       ", DEC1 distance
LOOP
```

Your Turn – Testing the Right IR LED/Detector Object Detector

Although there's some labeling involved, you can modify this program to test the right IR LED and detector by changing these two lines:

```
FREQOUT 8,1, irFrequency
irDetect = IN9
```

...so that they read:

```
FREQOUT 2,1, irFrequency
irDetect = IN0
```

- ✓ Modify TestLeftFrequencySweep.bs2 for testing the distance measurement of the right IR object detector.
- ✓ Run the program and verify that it can measure a similar distance.

Displaying Both Distances

It's useful at times to have a quick program you can run to test both the Boe-Bot's distance detectors at the same time. This program is organized into subroutines, which can be handy for copying and pasting into other programs that require distance detection.

Example Program – DisplayBothDistances.bs2

- ✓ Enter, save, and run DisplayBothDistances.bs2.
- ✓ Repeat the distance measurement exercise with a sheet of paper on each LED, then on both LEDs at the same time.

```
' -----[ Title ]-----------------------------------------------------------
' Robotics with the Boe-Bot - DisplayBothDistances.bs2
' Test IR detector distance responses of both IR object detectors to
' frequency sweep.

' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

' -----[ Variables ]-------------------------------------------------------

freqSelect     VAR     Nib
irFrequency    VAR     Word
irDetectLeft   VAR     Bit
irDetectRight  VAR     Bit
distanceLeft   VAR     Nib
distanceRight  VAR     Nib

' -----[ Initialization ]---------------------------------------------------

DEBUG CLS,
      "IR OBJECT ZONE", CR,
      "Left   Right", CR,
      "-----  -----"

' -----[ Main Routine ]-----------------------------------------------------

DO

  GOSUB Get_Distances
  GOSUB Display_Distances

LOOP
```

```
' -----[ Subroutine – Get_Distances ]----------------------------------------

Get_Distances:

  distanceLeft = 0
  distanceRight = 0

  FOR freqSelect = 0 TO 4

    LOOKUP freqSelect,[37500,38250,39500,40500,41500], irFrequency

    FREQOUT 8,1,irFrequency
    irDetectLeft = IN9
    distanceLeft = distanceLeft + irDetectLeft

    FREQOUT 2,1,irFrequency
    irDetectRight = IN0
    distanceRight = distanceRight + irDetectRight

    PAUSE 100

  NEXT

  RETURN

' -----[ Subroutine – Display_Distances ]-------------------------------------

Display_Distances:

  DEBUG CRSRXY,2,3, DEC1 distanceLeft,
        CRSRXY,9,3, DEC1 distanceRight
  RETURN
```

Your Turn – More Distance Tests

- ✓ Try measuring the distance of different objects and find out if the color and/or texture make any difference to the distance measurement.

ACTIVITY #2: BOE-BOT SHADOW VEHICLE

For one Boe-Bot to follow another, the Boe-Bot that follows, a.k.a. the shadow vehicle, has to know how far ahead the lead vehicle is. If the shadow vehicle is lagging behind, it has to detect this and speed up. If the shadow vehicle is too close to the lead vehicle, it has to detect this as well and slow down. If it's the right distance, it can wait until the measurements indicate it's too far or too close again.

Distance is just one kind of value that robots and other automated machinery are responsible for. When a machine is designed to automatically maintain a value, such as

distance, pressure, or fluid level, it generally involves a control system. These systems sometimes consist of sensors and valves, or sensors and motors, or, in the case of the Boe-Bot, sensors and continuous rotation servos. There is also some kind of processor that takes the sensor measurements and converts them to mechanical action. The processor has to be programmed to make decisions based on the sensor inputs, and then control the mechanical outputs accordingly. In the case of the Boe-Bot, the processor is the BASIC Stamp 2.

Closed loop control is a common method of maintaining levels, and it works very well for helping the Boe-Bot maintain its distance from an object. There are lots of different kinds of closed loop control. Some of the most common are hysteresis, proportional, integral, and derivative control. All of these types of control are introduced in detail in the Stamps in Class text Process Control, listed in the Preface.

Most control techniques can be implemented with just a few lines of code in PBASIC. In fact, the majority of the proportional control loop shown in Figure 8-4 reduces to just one line of PBASIC code. This diagram is called a block diagram, and it describes the steps of the proportional control process that the Boe-Bot will use to measure distance with its right IR LED and detector and adjust position to maintain distance with its right servo.

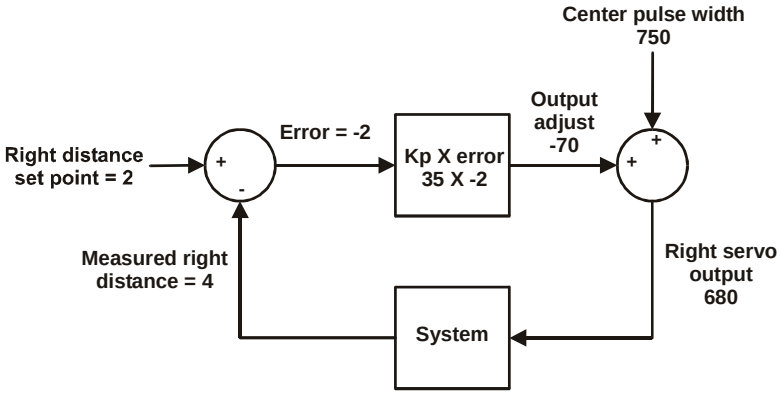


Figure 8-4
Proportional Control Block Diagram for Right Servo and IR Object Detector

Let's take a closer look at the numbers in Figure 8-4 to learn how proportional control works. This particular example is for the right IR LED/detector and right servo. The set point is 2, which means we want the Boe-Bot to maintain a distance of 2 between itself and any object it detects. The measured distance is 4, which is too far away. The error is the set point minus the measured distance which is 2 – 4 = -2. This is indicated by the symbols inside the circle on the left. This circle is called a summing junction. Next, the

error feeds into an operator block. This block shows that error will be multiplied by a value called a proportional constant (Kp). The value of Kp is 35. The block's output shows the result of -2 × 35 = -70, which is called the output adjust. This output adjust goes into another summing junction, and this time it is added to the servo's center pulse width of 750. The result is a 680 pulse width that will make the servo turn about ¾ speed clockwise. That makes the Boe-Bot's right wheel roll forward, toward the object. This correction goes into the overall system, which consists of the Boe-Bot, and the object, that was at a measured distance of 4.

The next time through the loop, the measured distance might change, but that's OK because 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, which is the difference between the set point and measured distances.

A control loop always has a set of equations that govern the system. The block diagram in Figure 8-4 is a way of visually describing this set of equations. Here are the equations that can be taken from this block diagram, along with solutions.

$$
\begin{aligned}
\textit{Error} &= \textit{Right distance set point} - \textit{Measured right distance} \\
&= 2 - 4 \\
\textit{Output adjust} &= \textit{error} \times K_p \\
&= -2 \times 35 \\
&= -70 \\
\textit{Right servo output} &= \textit{Output adjust} + \textit{Center pulse width} \\
&= -70 + 750 \\
&= 680
\end{aligned}
$$

By making some substitutions, the three equations above can be reduced to this one, which will give you the same result.

$$
\begin{aligned}
\textit{Right servo output} &= (\textit{Right distance set point} - \textit{Measured right distance}) \times Kp \\
&\quad + \textit{Center pulse width}
\end{aligned}
$$

By substituting the values from the example, we can see that the equation still works:

$$
\begin{aligned}
&= ((2 - 4) \times 35) + 750 \\
&= 680
\end{aligned}
$$

The left servo and IR object detector have a similar algorithm shown in Figure 8-5. The difference is that Kp is -35 instead of +35. Assuming the same measured value at the

right IR object detector, the output adjust results is a pulse width of 820. Here is the equation and calculations for this block diagram:

Left servo output = *(Left distance set point – Measured left distance) × Kp + Center pulse width*
= *((2 – 4) × –35) + 750*
= *820*

The result of this control loop is a pulse width that makes the left servo turn about ¾ of full speed counterclockwise. This is also a forward pulse for the left wheel. The idea of feedback is that the system's output is re-sampled, by the shadow Boe-Bot taking another distance measurement. Then the control loop repeats itself again and again and again…roughly 40 times per second.

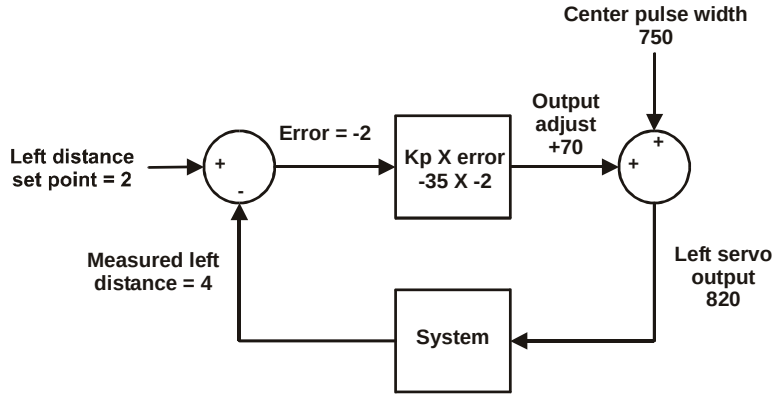


Figure 8-5
Proportional Control Block Diagram for Left Servo and IR Object Detector

Programming the Boe-Bot Shadow Vehicle

Remember that the equation for the right servo's output was:

Right servo output = *(Right distance set point – Measured right distance) × Kp + Center pulse width*

Here is an example of solving this same equation in PBASIC. The right distance set point is 2, the measured distance is a variable named **`distanceRight`** that will store the IR distance measurement, Kp is 35, and the center pulse width is 750:

```
pulseRight = 2 - distanceRight * 35 + 750
```

Remember that in PBASIC math expressions are executed from left to right. First, **`distanceRight`** is subtracted from 2. The result of this subtraction is then multiplied by **`Kpr`**, which is 35, and after that, the product is added to the center pulse width of 750.

You can use parentheses to force a calculation that is further to the right in a line of PBASIC code to be completed first. Recall this example: you can rewrite this line of PBASIC code:

```
pulseRight = 2 - distanceRight * 35 + 750
```

...like this:

```
pulseRight = 35 * (2 – distanceRight) + 750
```

In this expression, 35 is multiplied by the result of (2 – distanceRight), then the product is added to 750.

The left servo is different because Kp for that system is -35

```
pulseLeft = 2 - distanceLeft * (-35) + 750
```

Since the values -35, 35, 2, and 750 all have names, it's definitely a good place for some constant declarations.

```
Kpl            CON     -35
Kpr            CON     35
SetPoint       CON     2
CenterPulse    CON     750
```

With these constant declarations in the program, you can use the name **`Kpl`** in place of -35, **`Kpr`** in place of 35, **`SetPoint`** in place of 2, and **`CenterPulse`** in place of 750. After these constant declarations, the proportional control calculations now look like this:

```
pulseLeft  = SetPoint - distanceLeft  * Kpl + CenterPulse
pulseRight = SetPoint - distanceRight * Kpr + CenterPulse
```

The convenient thing about declaring constants for these values is that you can change them in one place, at the beginning of the program. The changes you make at the beginning of the program will be reflected everywhere these constants are used. For example, by changing the **`Kpl CON`** directive from -35 to -40, every instance of **`Kpl`** in the entire program changes from -35 to -40. This is exceedingly useful for experimenting with and tuning the right and left proportional control loops.

Example Program – FollowingBoeBot.bs2

FollowingBoeBot.bs2 repeats the proportional control loop just discussed with every servo pulse. In other words, before each pulse, the distance is measured and the error signal is determined. Then the error is multiplied by Kp, and the resulting value is added/subtracted to/from the pulse widths that are sent to the left/right servos.

- ✓ Enter, save, and run FollowingBoeBot.bs2.
- ✓ Point the Boe-Bot at an 8 ½ x 11" sheet of paper held in front of it as though it's a wall-obstacle. The Boe-Bot should maintain a fixed distance between itself and the sheet of paper.
- ✓ Try rotating the sheet of paper slightly. The Boe-Bot should rotate with it.
- ✓ Try using the sheet of paper to lead the Boe-Bot around. The Boe-Bot should follow it.
- ✓ Move the sheet of paper too close to the Boe-Bot, and it should back up, away from the paper.


```
' -----[ Title ]-----------------------------------------------------------
' Robotics with the Boe-Bot - FollowingBoeBot.bs2
' Boe-Bot adjusts its position to keep objects it detects in zone 2.

' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Constants ]-------------------------------------------------------

Kpl             CON     -35
Kpr             CON     35
SetPoint        CON     2
CenterPulse     CON     750

' -----[ Variables ]-------------------------------------------------------

freqSelect      VAR     Nib
irFrequency     VAR     Word
irDetectLeft    VAR     Bit
irDetectRight   VAR     Bit
distanceLeft    VAR     Nib
distanceRight   VAR     Nib
pulseLeft       VAR     Word
pulseRight      VAR     Word

' -----[ Initialization ]--------------------------------------------------

FREQOUT 4, 2000, 3000
```

```
' -----[ Main Routine ]-----------------------------------------------------

DO

  GOSUB Get_Ir_Distances

  ' Calculate proportional output.

  pulseLeft =  SetPoint - distanceLeft  * Kpl + CenterPulse
  pulseRight = SetPoint - distanceRight * Kpr + CenterPulse

  GOSUB Send_Pulse

LOOP

' -----[ Subroutine - Get IR Distances ]--------------------------------------

Get_Ir_Distances:
  distanceLeft = 0
  distanceRight = 0
  FOR freqSelect = 0 TO 4
    LOOKUP freqSelect,[37500,38250,39500,40500,41500], irFrequency

    FREQOUT 8,1,irFrequency
    irDetectLeft = IN9
    distanceLeft = distanceLeft + irDetectLeft

    FREQOUT 2,1,irFrequency
    irDetectRight = IN0
    distanceRight = distanceRight + irDetectRight
  NEXT
  RETURN

' -----[ Subroutine – Get Pulse ]---------------------------------------------

Send_Pulse:
  PULSOUT 13,pulseLeft
  PULSOUT 12,pulseRight
  PAUSE 5
  RETURN
```

How FollowingBoeBot.bs2 Works

FollowingBoeBot.bs2 declares four constants using the **`CON`** directive: **`Kpr`**, **`Kpl`**, **`SetPoint`**, and **`CenterPulse`**. Everywhere you see **`SetPoint`**, it's actually the number 2 (a constant). Likewise, everywhere you see **`Kpl`**, it's actually the number -35. **`Kpr`** is actually 35, and **`CenterPulse`** is 750.

```
Kpl             CON     -35
Kpr             CON     35
SetPoint        CON     2
CenterPulse     CON     750
```

The first thing the Main Routine does is call the **Get_Ir_Distances** subroutine. After the **Get_Ir_Distances** subroutine is finished, **distanceLeft** and **distanceRight** each contain a number corresponding to the zone in which an object was detected for both the left and right IR object detectors.

```
DO

  GOSUB Get_Ir_Distances
```

The next two lines of code implement the proportional control calculations for each servo.

```
  ' Calculate proportional output.

  pulseLeft =  SetPoint - distanceLeft  * Kpl + CenterPulse
  pulseRight = SetPoint - distanceRight * Kpr + CenterPulse
```

Now that the **pulseLeft** and **pulseRight** calculations are done, the **Send_Pulse** subroutine can be called.

```
  GOSUB Send_Pulse
```

The **LOOP** portion of the **DO…LOOP** sends the program back to the command immediately following the **DO** at the beginning of the main loop.

```
LOOP
```

Your Turn

Figure 8-6 shows a lead Boe-Bot followed by a shadow Boe-Bot. The lead Boe-Bot is running a modified version of FastIrRoaming.bs2, and the shadow Boe-Bot is running FollowingBoeBot.bs2. Proportional control makes the shadow Boe-Bot a very faithful follower. One lead Boe-Bot can string along a chain of 6 or 7 shadow Boe-Bots. Just add the lead Boe-Bot's side panels and tailgate to the rest of the shadow Boe-Bots in the chain.

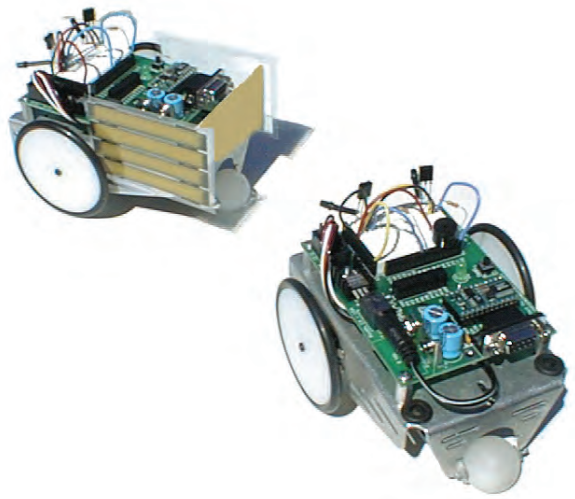


Figure 8-6
Lead Boe-Bot (left) and Shadow Boe-Bot (right)

- ✓ If you are part of a class, mount paper panels on the tail and both sides of the lead Boe-Bot as shown in Figure 8-6.
- ✓ If you are not part of a class (and only have one Boe-Bot) the shadow vehicle will follow a piece of paper or your hand just as well as it follows a lead Boe-Bot.
- ✓ Replace the 1 kΩ resistors that connect the lead Boe-Bot's P2 and P8 to the IR LEDs with 470 Ω or 220 Ω resistors.
- ✓ Program the lead Boe-Bot for object avoidance using a modified version of FastIrRoaming.bs2, renamed SlowerIrRoamingForLeadBoeBot.bs2.
- ✓ Make these modifications to SlowerIrRoamingForLeadBoeBot.bs2:
  - o Increase all **PULSOUT** ***Duration*** arguments that are now 650 to 710.
  - o Reduce all **PULSOUT** ***Duration*** arguments that are now 850 to 790.
- ✓ The shadow Boe-Bot should be running FollowingBoeBot.bs2 without any modifications.

- ✓ With both Boe-Bots running their respective programs, place the shadow Boe-Bot behind the lead Boe-Bot. The shadow Boe-Bot should follow at a fixed distance, so long as it is not distracted by another object such as a hand or a nearby wall.

You can adjust the set points and proportionality constants to change the shadow Boe-Bot's behavior. Use your hand or a piece of paper to lead the shadow Boe-Bot while doing these exercises:

- ✓ Try running FollowingBoeBot.bs2 using values of **`Kpr`** and **`Kpl`** constants, ranging from 15 to 50. Note the difference in how responsive the Boe-Bot is when following an object.
- ✓ Try making adjustments to the value of the SetPoint constant. Try values from 0 to 4.

ACTIVITY #3: FOLLOWING A STRIPE

Figure 8-7 shows an example of a course you can build and program your Boe-Bot to follow. Each stripe in this course is three long pieces of ¾ in (19 mm) vinyl electrical tape placed edge to edge on white poster board. No paper should be visible between the strips of electrical tape.

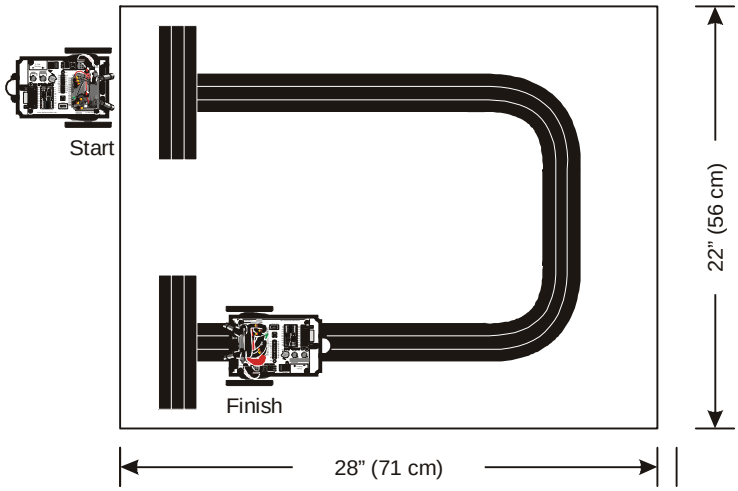


Figure 8-7
Stripe Following Course

Building and Testing the Course

For successful navigation of this course, some testing and Boe-Bot adjustment will be required.

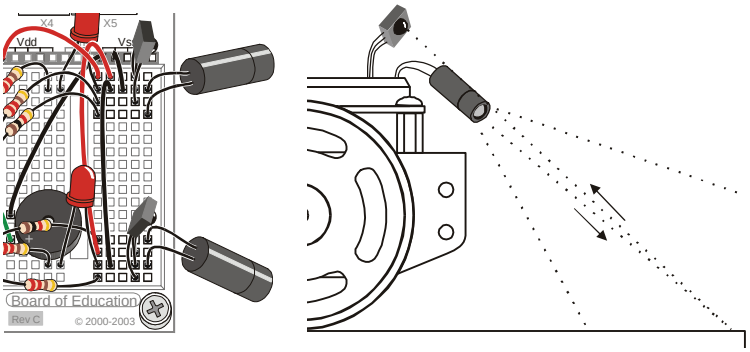
Materials Required

(1) Sheet of poster board, approximate dimensions: 22 X 28 in (56 X 71 cm)
(1) Roll of black vinyl electrical tape, ¾" (19 mm) wide

- ✓ On your poster board, use the electrical tape to lay out a course as shown in Figure 8-7.

Testing the Stripe

- ✓ Point your IR object detectors downward and outward as shown in Figure 8-8 (Figure 7-11 from page 242 repeated here for convenience).



Top View *Side View*

Figure 8-8
IR Object Detectors Directed Downwards to Scan for the Stripe

- ✓ Make sure your electrical tape course is free of fluorescent light interference. See Sniffing for IR Interference on page 233.
- ✓ Replace the 1 kΩ resistors in series with the IR LEDs with 2 kΩ resistors to make the Boe-Bot more nearsighted.
- ✓ Run DisplayBothDistances.bs2 from page 275. Keep your Boe-Bot connected to its programming cable so that you can see the displayed distances.
- ✓ Start by placing your Boe-Bot so that it is looking directly at the white background of your poster board as shown in Figure 8-9.

- ✓ Verify that your zone readings indicate that an object is detected in a very close zone. Both sensors should give you a 1 or 0 reading.

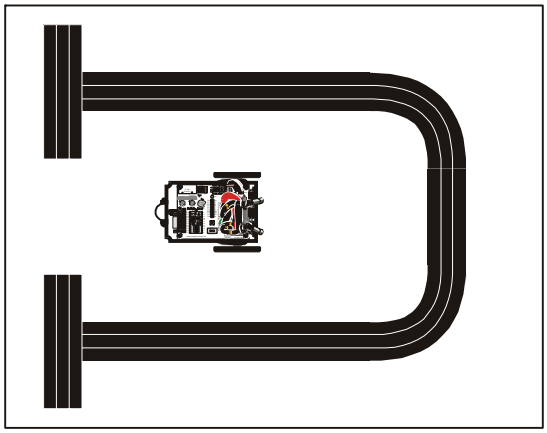


Figure 8-9
Test for Low Zone Number – Top View

- ✓ Place your Boe-Bot so that both IR object detectors are focused directly at the center of your electrical tape stripe (see Figure 8-10 and Figure 8-11).
- ✓ Then, adjust your Boe-Bot's position (toward and away from the tape) until both zone values reach the 4 or 5 level indicating that either a far away object is detected, or no object is detected.
- ✓ If you are having difficulties getting the higher readings with your electrical tape course, see Trouble Shooting the Electrical Tape Course on page 289.

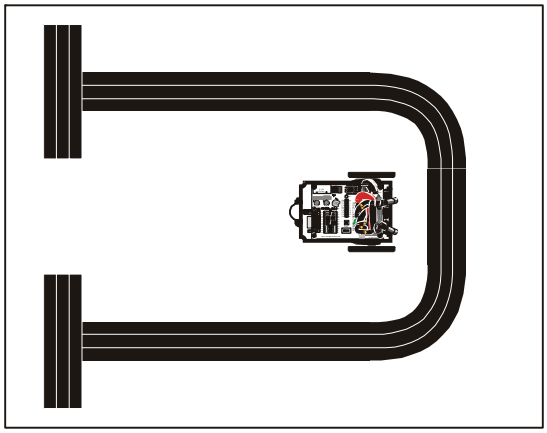


Figure 8-10
Test for High Zone Number – Top View

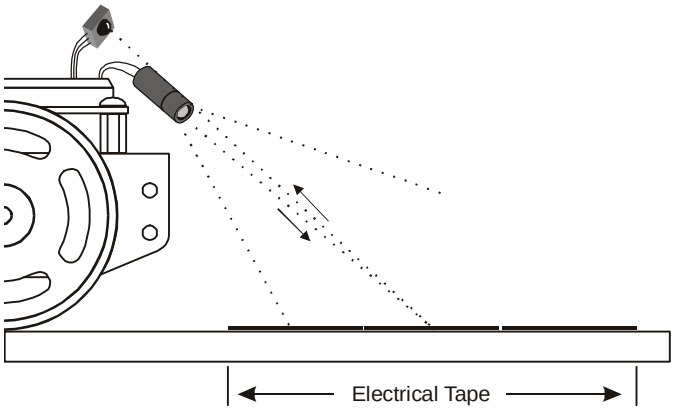


Figure 8-11
Test for High Zone Number – Side View

Troubleshooting the Electrical Tape Course

If you are unable to get a high zone value when the IR detectors are focused on the electrical tape, take a separate piece of paper, and make a stripe that's four strips wide instead of three. If the zone numbers are still low, make sure that you are using 2 kΩ resistors (red-black-red) in series with your IR LEDs. You can also try a 4.7 kΩ resistor to make the Boe-Bot more nearsighted. If none of this works, try a different brand of black vinyl electrical tape. Adjusting the IR LED/detector so that it is focused closer to or further from the front of the Boe-Bot (see Figure 8-11) may also help.

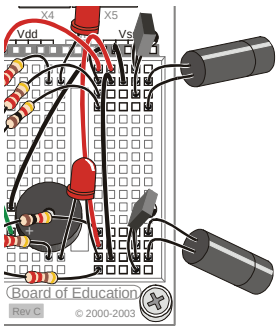
If you are having trouble with low zone measurements when reading the white surface, try pointing the IR LEDs and detectors further downward and toward the front of the Boe-Bot, but be careful not to cause reflection off the chassis. You can also try a lower-value resistor like 1 kΩ (brown-black-red).

- ✓ Now, place the Boe-Bot on the course so that its wheels straddle the black line. The IR detectors should be facing slightly outward. See close-up in Figure 8-12. Verify that the distance reading for both IR object detectors is 0 or 1 again. If the readings are higher, it means they need to be pointed slightly further outward, away from the edge of the stripe.

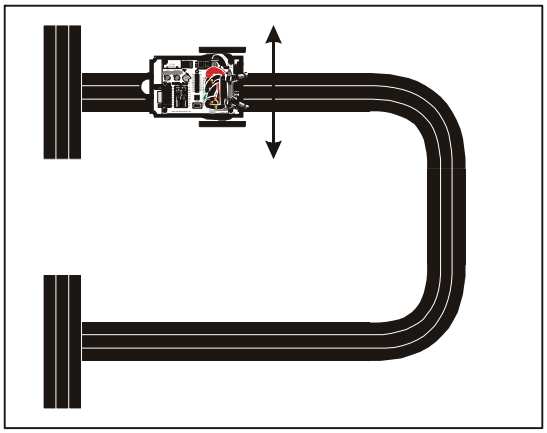
When you move the Boe-Bot in either direction indicated by the double-arrow, one or the other IR object detector will become focused on the electrical tape. When you do this, the readings for the object detector that is now over the electrical tape should increase to 4 or 5. Keep in mind that if you move the Boe-Bot toward its left, the right detectors should increase in value, and if you move the Boe-Bot toward its right, the left detectors should show the higher value.

- ✓ Adjust your IR object detectors until the Boe-Bot passes this last test. Then you will be ready to try following the stripe.

Figure 8-12: Stripe Scan Test



IR Object Detectors close-up



Top view of Boe-Bot straddling the stripe

Programming for Stripe Following

You will only need to make a few small adjustments to FollowingBoeBot.bs2 to make it work for following a stripe. First, the Boe-Bot should move toward objects closer than the **`SetPoint`** and away from objects further from the **`SetPoint`**. This is the opposite of how FollowingBoeBot.bs2 behaved. To reverse the direction the Boe-Bot moves when it senses that the object is not at the **`SetPoint`** distance, simply change the signs of **`Kpl`** and Kpr. In other words, change **`Kpl`** from -35 to 35, and change **`Kpr`** from 35 to -35. You will need to experiment with your **`SetPoint`**. Values from 2 to 4 tend to work best. This next example program will use a SetPoint of 3.

Example Program: StripeFollowingBoeBot.bs2

- ✓ Open FollowingBoeBot.bs2 and save it as StripeFollowingBoeBot.bs2.
- ✓ Change the **`SetPoint`** declaration from **`SetPoint CON 2`** to **`SetPoint CON 3`**.
- ✓ Change **`Kpl`** from -35 to 35.
- ✓ Change **`Kpr`** from 35 to -35.
- ✓ Run the program.

- ✓ Place your Boe-Bot at the "Start" location shown in Figure 8-13. The Boe-Bot should wait there until you place your hand in front of its IR object detectors. It will then roll forward. When it clears the starting stripe, take your hand away, and it should start tracking the stripe. When it sees the "Finish" stripe, it should stop and wait there.
- ✓ Assuming that you can get distance readings of 5 from the electrical tape and 0 from the poster board, **`SetPoint`** constant values of 2, 3, and 4 should work. Try different **`SetPoint`** values and make notes of your Boe-Bot's performance on the track.

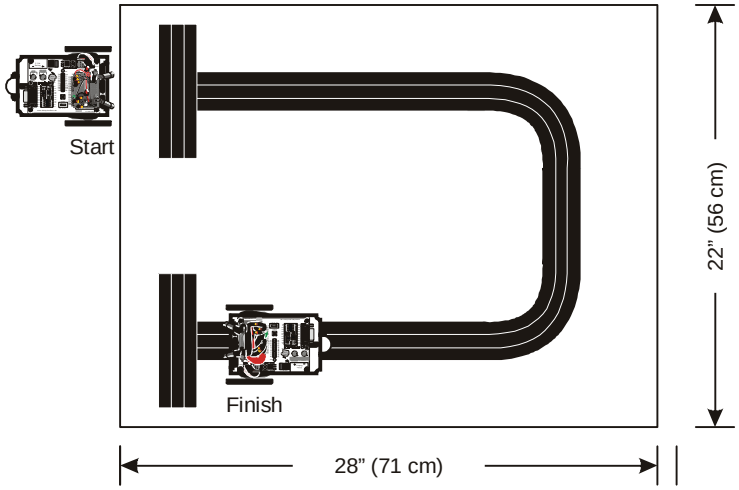


Figure 8-13
Stripe Following Course.

```
' -----[ Title ]-----------------------------------------------------------------
' Robotics with the Boe-Bot - StripeFollowingBoeBot.bs2
' Boe-Bot adjusts its position to move toward objects that are closer than
' zone 3 and away from objects further than zone 3.  Useful for following a
' 2.25 inch wide vinyl electrical tape stripe.
' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Constants ]-------------------------------------------------------

Kpl             CON     35                  ' Change from -35 to 35
Kpr             CON     -35                 ' Change from 35 to -35
SetPoint        CON     3                   ' Change from 2 to 3.
CenterPulse     CON     750
```

```
' -----[ Variables ]------------------------------------------------------

freqSelect     VAR     Nib
irFrequency    VAR     Word
irDetectLeft   VAR     Bit
irDetectRight  VAR     Bit
distanceLeft   VAR     Nib
distanceRight  VAR     Nib
pulseLeft      VAR     Word
pulseRight     VAR     Word

' -----[ Initialization ]--------------------------------------------------

FREQOUT 4, 2000, 3000

' -----[ Main Routine ]----------------------------------------------------

DO

  GOSUB Get_Ir_Distances

  ' Calculate proportional output.

  pulseLeft =  SetPoint - distanceLeft  * Kpl + CenterPulse
  pulseRight = SetPoint - distanceRight * Kpr + CenterPulse

  GOSUB Send_Pulse

LOOP

' -----[ Subroutine - Get IR Distances ]------------------------------------

Get_Ir_Distances:
  distanceLeft = 0
  distanceRight = 0
  FOR freqSelect = 0 TO 4
    LOOKUP freqSelect,[37500,38250,39500,40500,41500], irFrequency

    FREQOUT 8,1,irFrequency
    irDetectLeft = IN9
    distanceLeft = distanceLeft + irDetectLeft

    FREQOUT 2,1,irFrequency
    irDetectRight = IN0
    distanceRight = distanceRight + irDetectRight
  NEXT
  RETURN
```

```
' -----[ Subroutine - Get Pulse ]-----------------------------------------

Send_Pulse:
  PULSOUT 13,pulseLeft
  PULSOUT 12,pulseRight
  PAUSE 5
  RETURN
```

Your Turn – Stripe Following Contest

You can turn this into a contest with the lowest course time winning, provided the Boe-Bot faithfully waits at the "Start" and "Finish" stripes. You can make up other courses too. For best performance, experiment with different **`SetPoint`**, **`Kpl`**, and **`Kpr`** values.

ACTIVITY #4: MORE BOE-BOT ACTIVITIES AND PROJECTS ONLINE

So, what do you want to do with your Boe-Bot next? Possible next steps include:

- Projects with Boe-Bot accessories
- Contests and Challenges
- More activities with your Boe-Bot using the kit you've already got
- IR Remote for the Boe-Bot text and kit

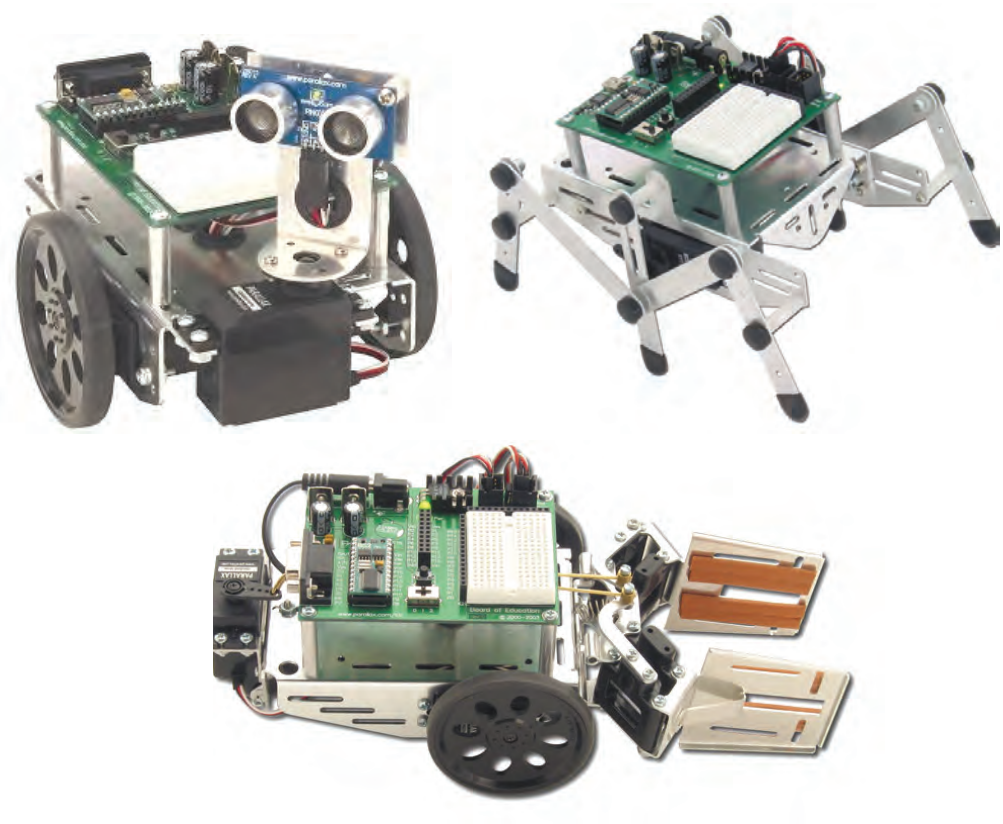
All of the resources discussed in this activity can be accessed through the www.parallax.com/go/Boe-Bot page.

Projects with Boe-Bot Accessories

Parallax has additional sensors and accessory kits so you can add capabilities and keep exploring with your Boe-Bot. Here are some examples:

- Ping))) Ultrasonic Distance Sensor (#28015) provides longer range and more accurate object distance measurements. An optional Mounting Bracket Kit (#570-28015) allows the sensor to sweep an area.
- Dual-axis accelerometer for tilt sensing (#28017)
- Compass Module for navigation (#29123)
- A Crawler Kit to make your Boe-Bot a 6-legged walker (#30055)
- A mechanical Gripper Kit for picking up and moving items (#28202)
- A Tank Tread Kit for all-terrain navigation (#28106)
- XBee RF modules and adapters for wireless control and communication (see www.parallax.com/go/XBee)

Figure 8-14: Ping))) Sensor and Bracket, Crawler Legs, and Gripper Add-Ons



Contests and Challenges

Interested in a contest? The www.parallax.com/go/Boe-Bot page also has links to rules to contests ranging from simple to complex and very challenging.

Some ideas are also included here in Appendix C: Boe-Bot Navigation Contests which begins on page 299.

IR Remote for the Boe-Bot

IR Remote for the Boe-Bot is available in print (#28139) and as a free PDF download. This book uses the same circuit you currently have built on your Boe-Bot, and has example programs that:

- Make it possible for you to drive your Boe-Bot around by pressing and holding certain buttons on the remote.
- Make your Boe-Bot to listen for configuration commands from the remote that tell it what to do next, like roam, follow objects, allow remote control, and more…

The only additional piece of equipment required is a universal TV remote, which is a common item in most households and can be obtained inexpensively through many stores as well as through www.parallax.com (#020-00001).

More Activities with Your Boe-Bot Using the Kit You've Already Got

Here are some examples of activities you can find at www.parallax.com/go/Boe-Bot that utilize the parts in your Boe-Bot kit. You won't need to buy any extra parts to try these activities:

- Better distance detection by varying IR LED brightness instead of frequency
- Navigate in a maze
- Detect a candle flame
- Climb uphill on a moving, tilting surface with an accelerometer

SUMMARY

Frequency sweep was introduced as a way of determining distance using the Boe-Bot's IR LED and detector. **FREQOUT** was used to send IR signals at frequencies ranging from 37.5 kHz (most sensitive) to 41.5 kHz (least sensitive). The distance was determined by tracking which frequencies caused the IR detector to report that an object was detected and which did not. Since not all of the frequencies were separated by the same value, the **LOOKUP** command was introduced as simple way to use the counting sequence supplied by a **FOR…NEXT** loop to index sequential lists of numbers.

Control systems were introduced along with closed loop control. Proportional control in a closed-loop system is an algorithm where the error is multiplied by a proportionality

constant to determine the system's output. The error is the measured system output subtracted from the set point. For the Boe-Bot, both system output and set point were in terms of distance. The BASIC stamp was programmed in PBASIC to operate control loops for the both the left and right servos and distance detectors. By re-sampling distance and adjusting the servo output before sending pulses to the servos, the control loop made the Boe-Bot responsive to object motion. The Boe-Bot was able to use proportional control to lock onto and follow objects, and it also used it to track and follow a stripe of black electrical tape.

Last but not least, pointers to more activities, resources and contests were covered since you're about done here.

Watch the Boe-Bot in Action at www.parallax.com!

You can see the Boe-Bot and other robot video clips in the Robot Videos section of the Video Gallery at www.parallax.com/go/videos.

Questions

1. What would the relative sensitivity of the IR detector be if you use **FREQOUT** to send a 35 kHz signal? What is the relative sensitivity with a 36 kHz signal?
2. Consider the code snippet below. If the **index** variable is 4, which number will be placed in the **prime** variable in this **LOOKUP** command? What values will **prime** store when index is 0, 1, 2, and 7?

```
LOOKUP index, [2, 3, 5, 7, 11, 13, 17, 19], prime
```

3. In what order are PBASIC math expressions evaluated? How can you override that order?
4. What PBASIC directive do you use to declare a constant? How would you give the number 100 the name "BoilingPoint?"

Exercises

1. List the sensitivity of the IR detector for each kHz frequency shown in Figure 8-1.
2. Write a segment of code that does the frequency sweep for just four frequencies instead of five.
3. Make a condensed checklist for the tests that should be performed to ensure faithful stripe following.

Projects

1. Create different types of electrical tape intersections and program the Boe-Bot to navigate through them. The intersections could be 90° left, 90° right, three-way, and four-way. This will involve the Boe-Bot recognizing it is at an intersection. When the Boe-Bot executes StripeFollowingBoeBot.bs2, the Boe-Bot will stay still at intersections. The goal is to have the Boe-Bot realize it's not doing anything and break from its proportional control loop.

   Hints: You can do this by creating two counters, one that increments by 1 each time through the **`DO…LOOP`**, and the other that only increments when the Boe-Bot delivers a forward pulse. When the counter that increments each time through the **`DO…LOOP`** gets to 60, use **`IF…THEN`** to check how many forward pulses were applied. If less than 30 forward pulses were applied, the Boe-Bot is probably stuck. Remember to reset both counters to zero each time the loop counter gets to 60. After the Boe-Bot recognizes that it is at an intersection, it needs to move to the top edge of the intersection, then back up and figure out whether it sees electrical tape or white background on the left and right, then make the correct 90° turn. Use a preprogrammed motion for turning 90°, without proportional control. For three-way and four-way intersections, the Boe-Bot may turn either right or left.
2. Advanced Optional Project - Design a maze-solving contest of your own, and program the Boe-Bot to solve it!

Solutions

Q1. The relative sensitivity at 35 kHz is 30%. For 36 kHz, it's 50%.

Q2. When **index** = 4, **prime** = 11.
- **index** = 0, **prime** = 2
- **index** = 1, **prime** = 3
- **index** = 2, **prime** = 5
- **index** = 7, **prime** = 19

Q3. Expressions are evaluated left to right. To override, use parentheses to change the order.

Q4. Use the **CON** directive.

```
BoilingPoint CON 100
```

E1. Frequency (kHz): 34 35 36 37 38 39 40 41 42
Sensitivity : 14% 30% 50% 76% 100% 80% 55% 35% 16%

E2. To solve this problem, put only four frequencies in the **LOOKUP** list, and decrease the **FOR…NEXT** index by one.

```
FOR freqSelect = 0 TO 3
  LOOKUP freqSelect, [37500, 38750, 39500, 40500], irFrequency
  FREQOUT 8, 1, irFrequency
  irDetect = IN9
  … commands not shown
NEXT
```

Add a **DEBUG** command to the **IF...THEN**. Don't forget the **ENDIF**.

```
  READ Dots + index, noteDot
  IF noteDot = 1 THEN
    noteDuration = noteDuration * 3 / 2
    DEBUG "Dotted Note!", CR
  ENDIF
```

E3. • Sniff for IR interference with IrInterferenceSniffer.bs2.
- Run Display BothDistances.bs2.
- White readings should be 0-1 in both sensors.
- Black readings should be 4-5 in both sensors.
- Straddle the line, both sensors should read 0-1.
- Move Boe-Bot back and forth over line, sensor over black line should read 4-5.

P1. In the solution below, the **Check_For_Intersection** subroutine implements the algorithm outlined. The left servo was arbitrarily chosen for counting the forward pulses. A bit-sized variable named **isStuck** is used as a flag to let the Main Routine know whether an intersection has been reached. In the **Navigate_Intersection** subroutine, the Boe-Bot goes forward past the intersection and then backs up, checking the sensors, using **DO…LOOP…UNTIL**. Then it makes a preprogrammed 90 degree turn in the correct direction. If the intersection is a 3-way or 4-way intersection, the Boe-Bot will arbitrarily turn in the direction that black is first detected. A constant, **Turn90Degree**, is provided to tune the 90 degree turn. Some audible and visual indicators are included, which aid in troubleshooting and understanding what the Boe-Bot is seeing and deciding, as well as adding a bit of personality and fun.

```
' -----[ Title ]-----------------------------------------------------
' Robotics with the Boe-Bot - IntersectionsBoeBot.bs2
' Navigate 90 degree left/right, 3-way, and 4-way intersections.
' Based on StripeFollowingBoeBot.bs2

' {$STAMP BS2}                    ' Stamp directive.
' {$PBASIC 2.5}                   ' PBASIC directive.
DEBUG "Program Running!"

' -----[ Constants ]-------------------------------------------------

Kpl             CON     35          ' Left proportional constant
Kpr             CON     -35         ' Right proportional constant
SetPoint        CON     3           ' 0-1 is White, 4-5 is Black
CenterPulse     CON     750
Turn90Degree    CON     30          ' Pulses needed for 90 turn

RightLED        PIN     1           ' LED Indicators
LeftLED         PIN     10

' -----[ Variables ]-------------------------------------------------

freqSelect      VAR     Nib         ' Sweep through 5 frequencies
irFrequency     VAR     Word        ' Freq sent to IR emitter
irDetectLeft    VAR     Bit         ' Store results from detectors
irDetectRight   VAR     Bit
distanceLeft    VAR     Nib         ' Calculate distance zones
distanceRight   VAR     Nib
pulseLeft       VAR     Word        ' Servo pulseWidths
pulseRight      VAR     Word
numPulses       VAR     Byte        ' Count total pulses
fwdPulses       VAR     Byte        ' Count forward pulses
counter         VAR     Byte
isStuck         VAR     Bit         ' Boolean variable,is bot stuck?
```

```
' -----[ Initialization ]-------------------------------------------

FREQOUT 4, 2000, 3000

' -----[ Main Routine ]----------------------------------------------

DO
  GOSUB Get_Ir_Distances           ' Read IR sensors
  GOSUB Update_LEDs                ' Indicate white/black line

' Calculate proportional output and move accordingly.
  pulseLeft =  SetPoint - distanceLeft  * Kpl + CenterPulse
  pulseRight = SetPoint - distanceRight * Kpr + CenterPulse
  GOSUB Send_Pulse

  GOSUB Check_For_Intersection     ' Are we stuck at intersection?
  IF (isStuck = 1) THEN
    GOSUB Make_Noise               ' Audible indication
    GOSUB Navigate_Intersection    ' Navigate through it
  ENDIF

LOOP

' -----[ Subroutines ]-----------------------------------------------

Navigate_Intersection:
'  Go forward until both sensors read white, through the intersection.
  DO
    pulseLeft = 850:  pulseRight = 650  ' Forward
    GOSUB Send_Pulse
    GOSUB Get_Ir_Distances
    GOSUB Update_LEDs
  LOOP UNTIL (distanceLeft <=2) AND (distanceRight <=2)

  GOSUB Stop_Quickly                    ' Don't coast forward

' Now back up until one detector sees the black.L & R turn will see
' black on one detector.3- or 4-way will see both black, turn toward
' whichever the bot sees first (random).
  DO
    pulseLeft = 650:  pulseRight = 850  ' Backward
    GOSUB Send_Pulse
    GOSUB Get_Ir_Distances
    GOSUB Update_LEDs
  LOOP UNTIL (distanceLeft >=4) OR (distanceRight >=4)

  GOSUB Stop_Quickly                    ' Don't coast backward

' Make 90 degree turn in direction of the detector which sees black
  IF (distanceLeft >=4) THEN           ' Left detector reads black
  FOR counter = 1 TO Turn90Degree      ' Turn 90 degrees left
```

```
    PULSOUT 13, 750                    ' without proportional control
    PULSOUT 12, 650
    PAUSE 20                           ' so use PAUSE 20
  NEXT
  ELSEIF (distanceRight >=4) THEN      ' Right detector reads black
   FOR counter = 1 TO Turn90Degree     ' Turn 90 degrees right
    PULSOUT 13, 850
    PULSOUT 12, 750
    PAUSE 20
  NEXT
  ENDIF

' That's it.  At this point the Boe-Bot should have turned 90 degrees
' to follow the intersection.  Continue following the black line.

  RETURN

Check_For_Intersection:
' Keep track of no. of pulses vs the forward pulses.  If there are less
' than 30 forward pulses per total of 60 pulses, robot is likely stuck
' at an intersection.

  isStuck = 0                          ' Initialize Boolean variable
  numPulses = numPulses + 1            ' Count total pulses sent

  SELECT numPulses
    CASE < 60
      IF (pulseLeft > CenterPulse) THEN
        fwdPulses = fwdPulses + 1      ' Count forward pulses
      ENDIF                            ' (forward is any pulse > 750)

    CASE = 60                          ' If we have sent 60 pulses
      IF (fwdPulses < 30) THEN         ' how many were forward?
        isStuck = 1                    ' If < 30, robot is stuck
      ENDIF

    CASE > 60
      numPulses = 0                    ' Reset counters back to zero
      fwdPulses = 0                    ' (Could reset in =60 case but
  ENDSELECT                            ' it spoils cool Make_Noise)
  RETURN

Make_Noise:
' Makes an increasing tone, proportional to number of forward pulses
  FOR counter = 1 TO fwdPulses STEP 3
    FREQOUT 4, 100, 3800 + (counter * 10)
  NEXT
  RETURN
```

```
Update_LEDs:
' Use LEDs to indicate whether detectors are seeing black or white.
' White = Off, Black = On.  Black is a distance reading > or = 4 .
  IF (distanceLeft >= 4)  THEN HIGH LeftLED  ELSE LOW LeftLED
  IF (distanceRight >= 4) THEN HIGH RightLED ELSE LOW RightLED
  RETURN

Stop_Quickly:
' This stops the wheels so the Boe-Bot does not "coast" forward.
  PULSOUT 13, 750
  PULSOUT 12, 750
  PAUSE 20
  RETURN

Get_Ir_Distances:
' Read both IR object detectors and calculate the distance.
' Black line gives 4-5 reading.  White surface give 0-1 reading.
  distanceLeft = 0
  distanceRight = 0
  FOR freqSelect = 0 TO 4
    LOOKUP freqSelect,[37500,38250,39500,40500,41500], irFrequency

    FREQOUT 8,1,irFrequency
    irDetectLeft = IN9
    distanceLeft = distanceLeft + irDetectLeft

    FREQOUT 2,1,irFrequency
    irDetectRight = IN0
    distanceRight = distanceRight + irDetectRight
  NEXT
  RETURN

Send_Pulse:
' Send a single pulse to the servos in between IR readings.
  PULSOUT 13,pulseLeft
  PULSOUT 12,pulseRight
  PAUSE 5                          ' PAUSE reduced due to IR readings
  RETURN
```

P2. If you create an interesting Boe-Bot maze project and you want to share it with others, you may want to join the Stamps in Class or Projects forums at http://forums.parallax.com. Or, you can email the Parallax Education Team directly at education@parallax.com.

A

Appendix A: Parts List and Kit Options

To complete the activities in this text, you will need a complete Boe-Bot robot and the electronic components necessary to build the example circuits. Kit options are described in this appendix. All of the information in this appendix was current at the time of printing. Parallax may make part substitutions at our discretion, out of necessity or to upgrade the quality of our products. For the latest information, downloads, and accessories, visit www.parallax.com/go/Boe-Bot.

Complete Boe-Bot Robot Kit Options

Aside from a PC with a serial or USB port and a few common household items, the Boe-Bot Robot Kit options contain all the parts and documentation you'll need to complete the experiments in this text.

Boe-Bot Robot Kit - Serial with USB Adapter (#28132) Parts and quantities subject to change without notice		
Stock Code	**Description**	**Quantity**
BS2-IC	BASIC Stamp 2 microcontroller module	1
28124	Robotics with the Boe-Bot Parts Kit	1
28125	Robotics with the Boe-Bot Student Guide	1
28150	Board of Education - Serial	1
700-00064	Parallax Screwdriver	1
800-00003	Serial cable	1
28031	USB to Serial Adapter and USB A to Mini B Cable	1

Boe-Bot Robot Kit - USB Only (#28832) Parts and quantities subject to change without notice		
Stock Code	**Description**	**Quantity**
BS2-IC	BASIC Stamp 2 microcontroller module	1
28124	Robotics with the Boe-Bot Parts Kit	1
28125	Robotics with the Boe-Bot Student Guide	1
28850	Board of Education USB	1
700-00064	Parallax Screwdriver	1
805-00006	USB A to Mini B Cable	1

Robotics with the Boe-Bot Parts kit

If you already have a Board of Education and BASIC Stamp 2, you may purchase the Robotics with the Boe-Bot Parts Kit, with or without this printed book:

Robotics with the Boe-Bot Parts & Text, #28154 Robotics with the Boe-Bot Parts only, #28124 Parts and quantities subject to change without notice		
Stock Code	**Description**	**Quantity**
150-01020	1 kΩ resistor	2
150-01030	10 kΩ resistor	4
150-02020	2 kΩ resistor	2
150-02210	220 Ω resistor	8
150-04710	470 Ω resistor	4
150-04720	4.7 kΩ resistor	2
200-01031	0.01 µF capacitor	2
200-01040	0.1 µF capacitor	2
350-00003	Infrared LED	2
350-00006	Red LED	2
350-00029	Phototransistor	2
350-00014	Infrared receiver (Panasonic PNA4602M or equivalent)	2
350-90000	LED standoff for infrared LED	2
350-90001	LED light shield for infrared LED	2
400-00002	Pushbutton, normally open	2
451-00303	3-Pin Header	2
700-00056	Whisker wire	2
800-00016	Jumper wires (bag of 10)	2
900-00001	Piezospeaker	1
	Boe-Bot Hardware Pack	1

Boe-Bot Hardware Pack Contents

Hardware replacement parts for the Boe-Bot can be purchased individually, as found in our on-line Robot Component Shop. Please note that the Hardware Pack is not sold as a unit separately from the Boe-Bot Robot (Full) Kits or the Boe-Bot Parts Kit.

Boe-Bot Hardware Pack Contents Parts and quantities subject to change without notice		
Parallax Stock Code	**Description**	**Quantity**
700-00002	Machine screw, 3/8” 4-40 pan-head, Phillips	8
700-00003	Hex nut, 4-40 zinc plated	10
700-00009	Tail wheel ball	1
700-00015	Nylon washer, #4 screw-size	2
700-00016	Machine screw, 4-40 x 3/8” flathead	2
700-00022	Boe-Bot aluminum chassis	1
700-00023	Cotter pin, 1/16" x 1.5” long	1
700-00025	Rubber grommet, 13/32"	2
700-00028	Machine screw, 4-40 x 1/ 4” pan-head Phillips	8
700-00038	Battery holder with cable and barrel plug	1
700-00060	Standoff, threaded aluminum, round 4-40	4
710-00007	Machine screw, 7/8” 4-40 pan-head, Phillips	2
713-00007	1/2” Spacer, aluminum, #4 round	2
721-00001	Parallax plastic wheel	2
721-00002	Rubber band tire	4
900-00008	Parallax Continuous Rotation Servo	2

Building a Boe-Bot with a BASIC Stamp HomeWork Board

The HomeWork Board, which is included in the BASIC Stamp Activity Kit (#90005), may be used with the Robotics with the Boe-Bot Parts kit and these additional items:

(2) 3-pin male/male headers, #451-00303
(1) Tinned-lead battery pack, #753-00001

A note to Educators: Quantity discounts are available for all of the kits listed above; see each kit’s product page at www.parallax.com for details. In addition, the BASIC Stamp HomeWork Board is available separately in packs of 10 as an economical solution for classroom use, costing significantly less than the Board of Education + BASIC Stamp 2 module (#28158). Contact the Parallax Sales Team toll free at (888) 512-1024.

Appendix B: Resistor Color Codes and Breadboarding Rules

B

RESISTOR COLOR CODES

Resistors like the ones we are using in this student guide have colored stripes that tell you what their resistance values are. There is a different color combination for each resistance value.

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.

Each color bar that tells you the resistor's value corresponds to a digit, and these colors/digits are listed in the table below. Figure B-1 shows how to use each color bar with the table to determine the value of a resistor.

Digit	Color
0	Black
1	Brown
2	Red
3	Orange
4	Yellow
5	Green
6	Blue
7	Violet
8	Gray
9	White

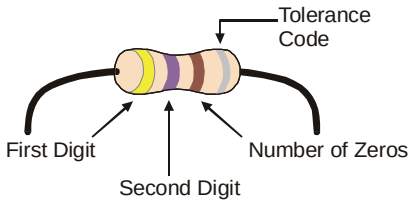


Figure B-1
Resistor Color Codes

- First stripe is yellow, which means leftmost digit is a 4.
- Second stripe is violet, which means next digit is a 7.
- Third stripe is brown. Since brown is 1, it means add one zero to the right of the first two digits.

The value of this resistor is 470 Ω.

BREADBOARDING RULES

Look at your Board of Education or HomeWork Board. The white square with lots of holes, or sockets, in it is called a solderless breadboard. This breadboard, combined with the black strips of sockets along two of its sides, is called the prototyping area (shown in Figure B-2).

The example circuits in this text are built by plugging components such as resistors, LEDs, speakers, and sensors into these small sockets. Components are connected to each other with the breadboard sockets. You will supply your circuit with electricity from the power terminals, which are the black sockets along the top labeled Vdd, Vin, and Vss. The black sockets on the left are labeled P0, P1, up through P15. These sockets allow you to connect your circuit to the BASIC Stamp's input/output pins.

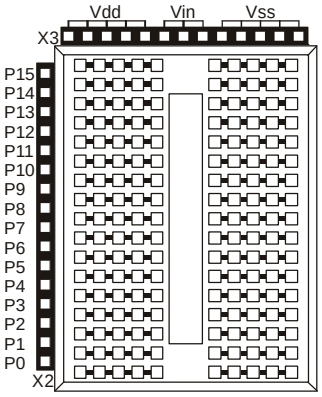


Figure B-2
Prototyping Area

Power terminals (black sockets along top), I/O pin access (black sockets along the side), and solderless breadboard (white sockets).

The breadboard has 17 rows of sockets separated into two columns by a trough. The trough splits each of the seventeen rows of sockets into two rows of five. Each row of five sockets is electrically connected inside the breadboard. You can use these rows of sockets to connect components together as dictated by a circuit schematic. If you insert two wires into any two sockets in the same 5-socket row, they are electrically connected to each other.

A circuit schematic is a roadmap that shows how to connect components together. It uses unique symbols each representing a different component. These component symbols are connected by lines to indicate an electrical connection. When two circuit symbols are connected by lines on a schematic, the line indicates that an electrical connection is made. Lines can also be used to connect components to voltage supplies. Vdd, Vin, and Vss all

have symbols. Vss corresponds to the negative terminal of the battery supply for the Board of Education or BASIC Stamp HomeWork Board. Vin is the battery's positive terminal, and Vdd is regulated to +5 volts.

Let's take a look at an example that uses a schematic to connect the parts shown in Figure B-3. For each of these parts, the part drawing is shown above the schematic symbol.

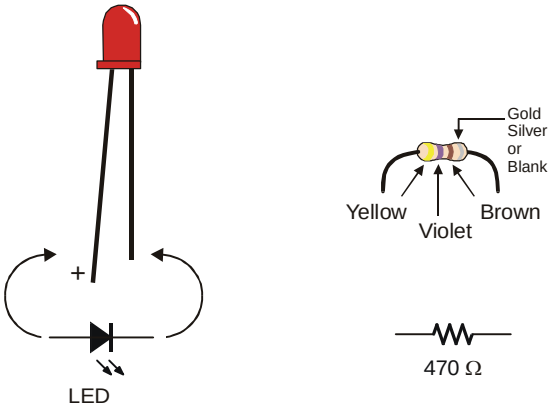


Figure B-3
Part Drawings and Schematic Symbols

LED(left) and
470 Ω resistor (right)

Figure B-4 shows an example of a circuit schematic on the left and a drawing of a circuit that can be built using this schematic on the right. Notice how the schematic shows that one end of the jagged line that denotes a resistor is connected to the symbol for Vdd. In the drawing, one of the resistor's two leads is plugged into one of the sockets labeled Vdd. In the schematic, the other terminal of the resistor symbol is connected by a line to the + terminal of the LED symbol. Remember, the line indicates the two parts are electrically connected. In the drawing, this is accomplished by plugging the other resistor lead into the same row of 5 sockets as the + lead on the LED. This electrically connects the two leads. The other terminal of the LED is shown connected to the Vss symbol in the schematic. In the drawing, the other lead of the LED is plugged into one of the sockets labeled Vss.

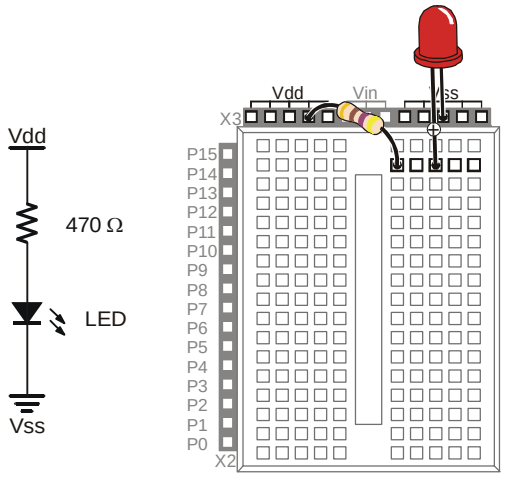


Figure B-4
Example Schematic and Wiring Diagram

Schematic (left) and wiring diagram (right)

Figure B-5 shows a second example of a schematic and wiring diagram. Here, P14 is connected to one end of a resistor, with the other end connected to the + terminal of an LED, and the – terminal of the LED is connected to Vss. These two schematics differ by only one connection. The resistor lead that used to be connected to Vdd is now connected to BASIC Stamp I/O pin P14. The schematic might look more different than that, because the resistor is shown drawn horizontally instead of vertically. But in terms of connections, it only differs by one, P14 in place of Vdd.

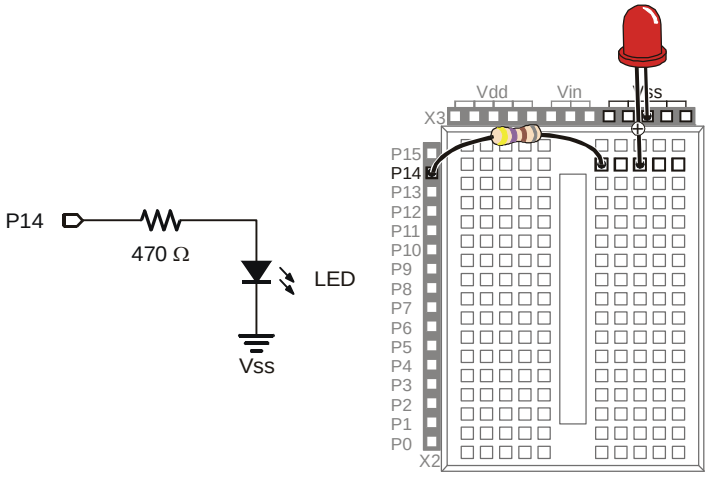


Figure B-5
Example Schematic and Wiring Diagram

Schematic (left) and wiring diagram (right)

Here is a more complex example that involves two additional parts, a 1 kΩ resistor, a phototransistor, and a capacitor. The schematic symbols and part drawings for the components you are not already familiar with are shown in Figure B-6. The phototransistor's terminals are labeled C, B, and E. The B terminal is optical, so it doesn't have any electrical connections. The C terminal is the longer pin, and the E terminal is the shorter pin that comes out of the plastic enclosure closer to a flat spot on its side.

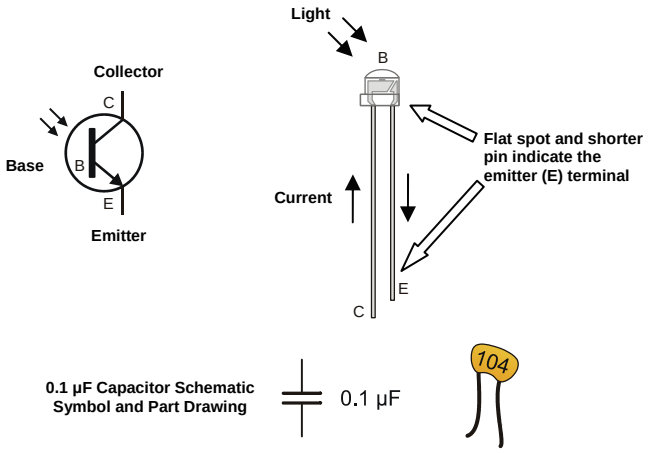


Figure B-6
Part Drawings and Schematic Symbols

Phototransistor (top)
Non-polar capacitor (bottom)

Since this schematic shown in Figure B-7 calls for a 1 kΩ resistor, which is 1000 Ω, the first step is to consult Appendix C: Resistor Color Codes to determine the color code. The color code is Brown, Black, Red. This resistor is connected to P6 in the schematic, which corresponds to the resistor lead plugged into the socket labeled P6 in the prototyping area (Figure B-8). In the schematic, the other lead of the resistor is connected to not one, but two other component terminals: the phototransistor's C terminal and one of the capacitor's terminals. On the breadboard, that resistor lead is plugged into one of the of the breadboard's 5-socket rows. This row also has the phototransistor's C lead and one of the capacitor's leads plugged into it. In the schematic, the phototransistor's E terminal and the capacitor's other lead are connected to Vss. Here is a trick to keep in mind when building circuits on a breadboard. You can use a wire to connect an entire row on the breadboard to another row, or even to I/O pins or power terminals such as Vdd or Vss. In this case, a wire was used to connect Vss to a row on the breadboard. Then, the phototransistor's E lead and the capacitor's other lead are plugged into the same row, which connects them to Vss, completing the circuit.

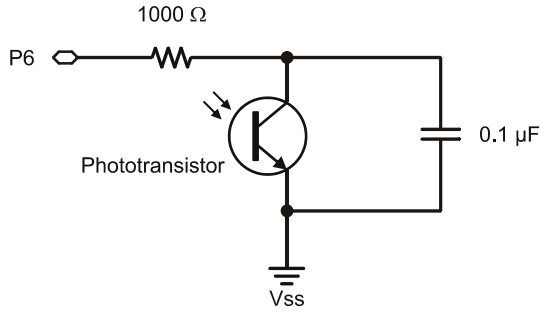


Figure B-7
Resistor, Phototransistor, and Capacitor Schematic

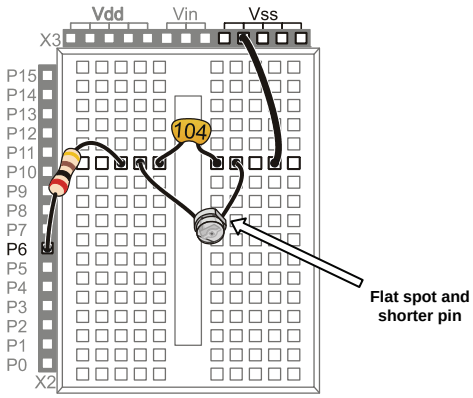


Figure B-8
Resistor, Phototransistor, and Capacitor Wiring Diagram

Keep in mind that the wiring diagrams presented here as solutions to the schematics are not the ONLY solutions to those schematics. For example, Figure B-9 shows another solution to the schematic just discussed. Follow the connections and convince yourself that it does satisfy the schematic.

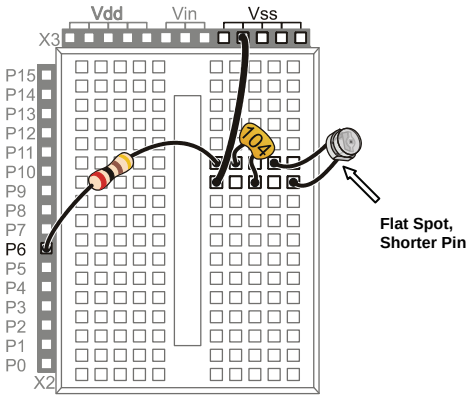


Figure B-9
Resistor, Phototransistor, and Capacitor Wiring Diagram

Note the alternative parts placement.

Appendix C: Boe-Bot Navigation Contests

If you're planning a competition for autonomous robots, these rules are provided courtesy of Seattle Robotics Society.

CONTEST #1: ROBOT FLOOR EXERCISE

Purpose

The floor exercise competition is intended to give robot inventors an opportunity to show off their robots or other technical contraptions.

Rules

The rules for this competition are quite simple. A 10-foot-by-10-foot flat area is identified, preferably with some physical boundary. Each contestant will be given a maximum of five minutes in this area to show off what their robot can do. The robot's contestant can talk through the various capabilities and features of the robot. As always, any robot that could damage the area or pose a danger to the public will not be allowed. Robots need not be autonomous, but it is encouraged. Judging will be determined by the audience, either indicated by clapping (the loudest determined by the judge), or some other voting mechanism.

CONTEST #2: LINE FOLLOWING

Objective

To build an autonomous robot that begins in Area "A" (at position "S"), travels to Area "B" (completely via the line), then travels to the Area "C" (completely via the line), then returns to the Area "A" (at position "F"). The robot that does this in the least amount of time (including bonuses) wins. The robot must enter areas "B" and "C" to qualify. The exact layout of the course will not be known until contest day, but it will have the three areas previously described.

Skills Tested

The ability to recognize a navigational aid (the line) and use it to reach the goal.

Maximum Time to Complete Course

Four minutes.

Example Course

All measurements in the example course are approximate. There is a solid line dividing Area "A" from Area "T" at position "F." This indicates where the course ends. The line is black, approximately 2.25 inches wide and spaced approximately two feet from the walls. All curves have a radius of at least one foot and at most three feet. The walls are 3 1/2 inches high and surround the course. The floor is white and made of either paper or Dupont Tyvek®. Tyvek is a strong plastic used in mailing envelopes and house construction.

Positions "S" and "F" are merely for illustration and are not precise locations. A Competitor may place the robot anywhere in Area "A," facing in any direction when starting. The robot must be completely within Area "A." Areas "A," "B" and "C" are not colored red on the actual course.

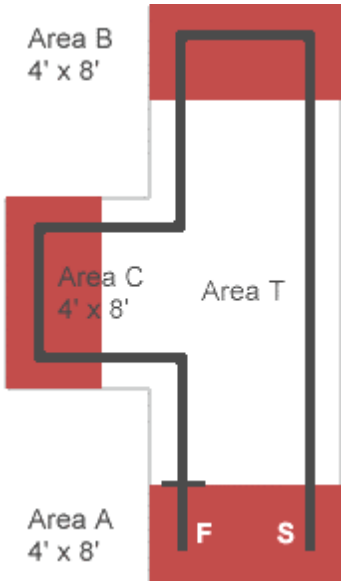


Figure D-1
Sample Contest Course

Scoring

Each contestant's score is calculated by taking the time needed to complete the course (in seconds) minus 10% for each "accomplishment." The contestant with the lowest score wins.

Line Following Scoring	
Accomplished	**Percent Deducted**
Stops in area A after reaching B and C	10%
Does not touch any walls	10%
Starts on command	10%

("Starts on command" means the robot starts with an external, non-tactile command. This could, for example, be a sound or light command.)

CONTEST #3: MAZE FOLLOWING

Purpose

The grand maze is intended to present a test of navigational skills by an autonomous robot. The scoring is done in such a way as to favor robots which are either brutally fast or which can learn the maze after one pass. The object is for a robot, which is set down at the entrance of the maze, to find its way through the maze and reach the exit in the least amount of time.

Physical Characteristics

The maze is constructed of 3/4" shop-grade plywood. The walls are approximately 24 inches high, and are painted in primary colors with glossy paint. The walls are set on a grid with 24-inch spacing. Due to the thickness of the plywood and limitations in accuracy, the hallways may be as narrow as 22 inches. The maze can be up to 20-feet square, but may be smaller, depending on the space available for the event.

The maze will be set up on either industrial-type carpet or hard floor (depending on where the event is held). The maze will be under cover, so your robot does not have to be rain proof; however, it may be exposed to various temperatures, wind, and lighting conditions. The maze is a classical two-dimensional proper maze: there is a single path from the start to the finish and there are no islands in the maze. Both the entrance and exit are located on outside walls. Proper mazes can be solved by following either the left wall

or the right wall. The maze is carefully designed so that there is no advantage if you follow the left wall or the right wall.

Robot Limitations

The main limit on the robot is that it be autonomous: once started by the owner or handler, no interaction is allowed until the robot emerges from the exit, or it becomes hopelessly stuck. Obviously the robot needs to be small enough to fit within the walls of the maze. It may touch the walls, but may not move the walls to its advantage -no bulldozers. The judges may disqualify a robot which appears to be moving the walls excessively. The robot must not damage either the walls of the maze, nor the floor. Any form of power is allowed as long as local laws do not require hearing protection in its presence or place any other limitations on it.

Scoring

Each robot is to be run through the maze three times. The robot with the lowest single time is the winner. The maximum time allowed per run is 10 minutes. If a robot cannot finish in that amount of time, the run is stopped and the robot receives a time of 10 minutes. If no robot succeeds in finding the exit of the maze, the one that made it the farthest will be declared the winner, as determined by the contest's judge.

Logistics

Each robot will make one run, proceeding until all robots have attempted the maze. Each robot then does a second run through the maze, then the robots all do the third run. The judge will allow some discretion if a contestant must delay their run due to technical difficulties. A robot may remember what it found on a previous run to try to improve its time (mapping the maze on the first run), and can use this information in subsequent runs-as long as the robot does this itself. It is not allowed to manually "configure" the robot through hardware or software as to the layout of the maze.

Index

Parts and quantities are subject to change without notice. Parts may differ from what is shown in this picture. If you have any questions about your kit, please email stampsinclass@parallax.com.