Robotics with the Boe-Bot

Student Guide

VERSION 2.2

PARALLAX INC

WARRANTY

Parallax Inc. 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 Inc. 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 Inc. 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 1-928982-03-4

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 Inc. 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.

WEB SITE AND DISCUSSION LISTS

The Parallax Inc. web site (www.parallax.com) has many downloads, products, customer applications and on-line ordering for the components used in this text. We also maintain several e-mail discussion lists for people interested in using Parallax products. These lists are accessible from www.parallax.com via the Support → Discussion Groups menu. These are the lists that we operate:

- BASIC Stamps – This list is widely utilized by engineers, hobbyists and students who share their BASIC Stamp projects and ask questions.
- Stamps in Class – Created for educators *and* students, subscribers discuss the use of the Stamps in Class curriculum in their courses. The list provides an opportunity for both students and educators to ask questions and get answers.
- Parallax Educators –Exclusively for educators and those who contribute to the development of Stamps in Class. Parallax created this group to obtain feedback on our curricula and to provide a forum for educators to develop and obtain Teacher's Guides.
- Parallax Translators – The purpose of this list is to provide a conduit between Parallax and those who translate our documentation to languages other than English. Parallax provides editable Word documents to our translating partners and attempts to time the translations to coordinate with our publications.
- Toddler Robot – A customer created this discussion list to discuss applications and programming of the Parallax Toddler robot.
- SX Tech – Discussion of programming the SX microcontroller with Parallax assembly language tools and 3rd party BASIC and C compilers.
- Javelin Stamp – Discussion of application and design using the Javelin Stamp, a Parallax module that is programmed using a subset of Sun Microsystems' Java® programming language.

ERRATA

While great effort is made to assure the accuracy of our texts, errors may still exist. If you find an error, please let us know by sending an email to editor@parallax.com. We continually strive to improve all of our educational materials and documentation, and frequently revise our texts. Occasionally, an errata sheet with a list of known errors and corrections for a given text will be posted to our web site, www.parallax.com. Please check the individual product page's free downloads for an errata file.

Table of Contents

Preface

FOREWORD

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 difference 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 in Figure P-1.

Figure P-1
Parallax Inc’s Boe-Bot™
Autonomous Wheeled 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.

AUDIENCE

The *Robotics with the Boe-Bot* Student Guide was created for ages 13+ as a subsequent text to *"What's a Microcontroller?"*. Like all of the Stamps in Class curriculum, this series of experiments teaches new techniques and circuits with minimal overlap between the other texts. The general topics introduced in this series are: basic Boe-Bot navigation under program control, navigation using a variety of sensor inputs, navigation using feedback and various control techniques, and navigation using programmed artificial intelligence. Each topic is addressed in an introductory format designed to impart a conceptual understanding along with some hands-on experience. Those who intend to delve further into industrial technology, electronics, or robotics are likely to benefit significantly from initial experiences with these topics.

SUPPORT AND DISCUSSION GROUPS

The following two Yahoo! Discussion Groups are available for those who would like support in using this text. These groups are accessible from www.parallax.com under Discussion Groups on the Support menu.

Stamps In Class Group: Open to students, educators, and independent learners, this forum allows members to ask each other questions and share answers as they work through the activities, exercises and projects in this text.

Parallax Educator's Group: This moderated forum provides support for educators and welcomes feedback as we continue to develop our Stamps in Class curriculum. To join this group you must have proof of your status as an educator verified by Parallax. The Teacher's Guide for this text is available as a free download through this forum.

Educational Support: stampsinclass@parallax.com Contact the Parallax Stamps in Class Team directly if you are having difficulty subscribing to either of these Yahoo! Groups, or have questions about the material in this text, our Stamps in Class Curriculum, our Educator's Courses, or any of our educational services.

Educational Sales: sales@parallax.com Contact our Sales Team for information about educational discount pricing and classroom packs for our Stamps in Class kits and other selected products.

Technical Support: support@parallax.com Contact our Tech Support Team for general questions regarding the set-up and use of any of our hardware or software products.

THE STAMPS IN CLASS CURRICULUM

This text can be successfully completed with no prerequisites. However, *What's a Microcontroller?* is the recommended first (gateway) text to our Stamps in Class curriculum.

"*What's a Microcontroller?*", Student Guide, Version 2.2, Parallax Inc., 2004

After completing this text, you can continue your studies with any of the kits and student guides or other manuals discussed below. All of these publications are available for free download from www.parallax.com. The versions cited below were current at the time of this printing. We continually strive to improve our educational program. Please check our web sites, www.parallax.com and www.stampsinclass.com, for the latest revisions.

Stamps in Class Student Guides:

For a well-rounded introduction to the design practices that go into modern devices and machinery, working through the activities and projects in the following Student Guides is highly recommended.

"*Applied Sensors*", Student Guide, Version 1.3, Parallax Inc., 2003
"*Basic Analog and Digital*", Student Guide, Version 1.3, Parallax Inc., 2004
"*Process Control*", Student Guide, Version 2.0, Parallax Inc., 2004

More Robotics Kits:

After completing this text, you will be ready for either or both of these more advanced robotics texts and kits:

"*Advanced Robotics: with the Toddler*", Student Guide, Version 1.2, Parallax Inc., 2003
"*SumoBot*" Manual, Version 2.0, Parallax Inc., 2004

Educational Project Kits:

Elements of Digital Logic, Understanding Signals and *Experiments with Renewable Energy* focus more closely on topics in electronics, while *StampWorks* provides a variety of projects that are useful to hobbyists, inventors and product designers interested in trying a variety of projects.

"*Elements of Digital Logic*", Student Guide, Version 1.0, Parallax Inc., 2003
"*Experiments with Renewable Energy*", Student Guide, Version 1.0, Parallax Inc., 2004
"*StampWorks*", Manual, Version 1.2, Parallax Inc., 2000-2001
"*Understanding Signals*", Student Guide, Version 1.0, Parallax Inc., 2003

Reference

The *BASIC Stamp Manual* is an essential reference for all Stamps in Class Student Guides. It is packed with information on the BASIC Stamp microcontrollers, the BASIC Stamp Editor, and the PBASIC programming language.

"*BASIC Stamp Manual*", Version 2.0c, Parallax Inc., 2000

FOREIGN TRANSLATIONS

Parallax educational texts may be translated to other languages with our permission (e-mail stampsinclass@parallax.com). If you plan on doing any translations please contact us so we can provide the correctly-formatted MS Word documents, images, etc. We also maintain a discussion group for Parallax translators that you may join. It's called the Parallax Translators Yahoo! Group, and directions for finding it are included on the inside cover of this text. See section entitled: WEB SITE AND DISCUSSION LISTS on the page before the Table of Contents.

SPECIAL CONTRIBUTORS

Chuck Schoeffler, Ph.D., authored portions of the v1.2 text in conjunction with Parallax, Inc. At that time, Dr. Schoeffler was a professor at University of Idaho's Industrial Technology Education department. He designed the original Board of Education Robot (Boe-Bot) shown in Figure P-2 along with similar robot derivatives with many unique functions. After several revisions, Chuck's design was adopted as the basis of the Parallax Boe-Bot that is used in this text. Russ Miller of Parallax designed the Boe-Bot based on this prototype.

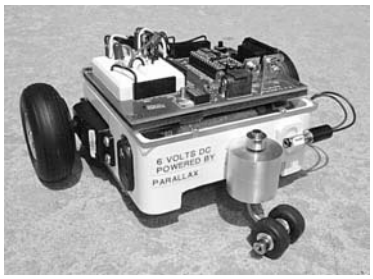


Figure P-2
Original Boe-Bot Prototype

Andrew Lindsay, Parallax Chief Roboticist, has since rewritten this text and its activities with three goals in mind. First, to support all activities in the text with carefully written "how to" instructions. Second, to expose the reader and student to new circuit, programming, engineering, and robotic concepts in each chapter. Third, to ensure that the experiments can be performed with a high degree of success using the most up-to-date Parallax equipment. As of this version, the most up-to-date equipment is the Board of Education Rev C or the BASIC Stamp HomeWork Board.

Thanks to Dale Kretzer for editorial review, which was incorporated into v1.4. Thanks also to the following Stamps in Class e-group participants for their input: Richard Breen, Robert Ang, Dwayne Tunnell, Marc Pierloz, and Nagi Babu. These participants submitted one or more of the following: error corrections, useful editorial suggestions, or new material for v1.4. Thanks to student Laura Wong and to Rob Gerber for their respective contributions to v1.5. A special thanks to the Parallax, Inc. staff. Each and every member of the Parallax team has in some way contributed to making the Stamps in Class program a success.

Version 2.0 of this Student Guide was a major revision and complete rewrite, featuring new activities, PBASIC 2.5 support, and BASIC Stamp HomeWork Board support. This revision would not have been possible without the following people. Parallaxians: Andy Lindsay – author, Rich Allred – technical illustration, Stephanie Lindsay – technical editing, Kris Magri – reviewer and robotics guru. Also, thanks go to Stamps in Class outside reviewers and contributors Robert Ang and Sid Weaver.

If you have suggestions, think you found a mistake, or would like to contribute an activity or chapter to forthcoming *Robotics with the Boe-Bot* versions or *More Robotics with the Boe-Bot* texts, contact us at stampsinclass@parallax.com. Subscribe and stay tuned to the StampsInClass Yahoo! Group for the latest in free hardware offers for *Robotics with the Boe-Bot* contributions. See the WEB SITE AND DISCUSSION LISTS section on the page before the Table of Contents.

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.

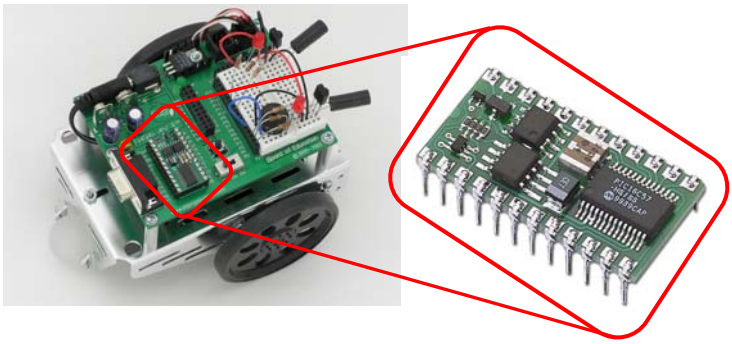


Figure 1-1 BASIC Stamp® 2 module on a Boe-Bot™ robot.

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 makes BASIC Stamp microcontrollers.
- Beginners - Made for beginners to use 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 instruct a computer how to solve problems
- Code - In terms that you and the computer 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, and it's also included on the Parallax CD. Many electronics outlets carry the What's a Microcontroller Kits and printed Student Guides. If you have any difficulty finding them locally, they can also be purchased directly from Parallax, either on-line at www.parallax.com or by phone at (888) 512-1024.

HARDWARE AND SOFTWARE

Getting started with BASIC Stamp programming is similar to getting started with a brand-new PC or laptop. The first things that most people do when they get a new PC or laptop 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 BASIC Stamp microcontrollers, you will be doing all these same activities. This chapter shows you how to get up and running with BASIC Stamp programming as it guides you through:

- Finding and installing the programming software
- Connecting your BASIC Stamp module to a battery power supply
- Connecting your BASIC Stamp module to the computer for programming
- Writing your first few PBASIC programs
- Disconnecting power when you're done

If you are in a class, the BASIC Stamp may already be all set up for you. If this is the case, your teacher may have other instructions. If not, the activities in this chapter will take you through all the steps of getting your new BASIC Stamp microcontroller up and running.

√ If you have already completed the *What's a Microcontroller?* Student Guide, skip to the next chapter.

√ Likewise, if you are already familiar with your BASIC Stamp and Board of Education or BASIC Stamp HomeWork Board, skip to the next chapter.

Both this text and *What's a Microcontroller?* contain instructions for getting started with BASIC Stamp hardware and software in Chapter 1. These instructions are almost identical.

Introducing the BASIC Stamp and Board of Education

A BASIC Stamp 2 module and a Board of Education® carrier board are shown in Figure 1-2. As mentioned earlier, a BASIC Stamp module is like a very small computer. This very small computer plugs into the Board of Education carrier board. As you will soon see, the Board of Education makes it easy to connect a power supply and serial cable to the BASIC Stamp module. In later activities, you will also see how the Board of Education makes it easy to build circuits and connect them to the BASIC Stamp.

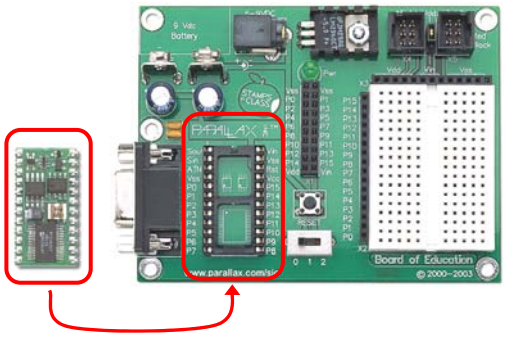


Figure 1-2
BASIC Stamp® 2 Module (left)

Board of Education® Carrier Board (right)

Introducing the BASIC Stamp HomeWork Board

The BASIC Stamp® HomeWork Board™ project platform is shown below in Figure 1-3. This board is like a Board of Education with the BASIC Stamp 2 microcontroller built in. You can use either a BASIC Stamp 2 module with Board of Education carrier board or the BASIC Stamp HomeWork Board as your project platform for the activities in this text. Be sure to follow the directions for the specific project platform you are using, since they differ in a few places.

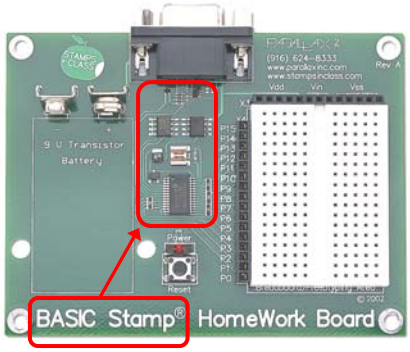


Figure 1-3
BASIC Stamp®
HomeWork Board™
project platform.

What's the difference?

Using a Board of Education carrier board and BASIC Stamp module gives you additional features such as headers for plugging in servo motors, control over the type of power supply the servos receive, and a handy, 3-position switch you can use to control what parts of the system receive power. The BASIC Stamp 2 module is removable, and can be replaced.

The BASIC Stamp HomeWork Board has no servo ports, external power supply jack or power switch, but it also costs less. You have to build your own servo connections, and to control power by disconnecting it from the board, or by building your own power control circuits. The BASIC Stamp 2 microcontroller is build right into the board, and each I/O pin is protected by a surface-mounted 220 Ω resistor.

See also: Appendix B: BASIC Stamp and Carrier Board Components and Features

ACTIVITY #1: GETTING THE SOFTWARE

The BASIC Stamp Editor (version 2.0 or higher) is the software you will use in most of the activities and projects in this text. This software allows you to write programs on your computer and download them into your Boe-Bot's BASIC Stamp brain. It also displays messages on your computer screen sent by the BASIC Stamp, allowing your Boe-Bot one way to report what it is doing and sensing to you, the roboticist.

The **BASIC Stamp Editor is free software**, and the two easiest ways to get it are:

- Download from the Internet: Look for "BASIC Stamp Windows Editor Version 2.0…" on the www.parallax.com → Downloads → BASIC Stamp Software page.
- Included on the Parallax CD: Follow the Software link on the Welcome page. Make sure the date printed on the CD is more recent than April 2003.

In a Hurry? Get your copy of the BASIC Stamp Windows Editor version 2.0 (or higher) and install it on your PC or laptop. Then, skip to: Activity #3: Setting up the Hardware and Testing the System.

If you have questions along the way, Activity #1 can be used as a step-by-step reference for getting the software, and Activity #2 can be used as a reference for installing the software on your PC or laptop.

Computer System Requirements

You will need either a PC or laptop computer to run the BASIC Stamp Editor software. Getting started with the BASIC Stamp is easiest if your PC or laptop has the following features:

- Windows 98 or newer operating system
- A serial or USB port
- A CD-ROM drive, World Wide Web access, or both

USB Port Adapter: If your computer only has USB ports, you will need a USB to Serial Adapter. See the information box on page 14 for details.

Downloading the Software from the Internet

It's easy to download the BASIC Stamp Editor software from the Parallax web site. The web page shown in Figure 1-4 may look different from the web page you see when you visit the site. Nonetheless, the steps for downloading the software should still be similar to these:

√ Using a web browser, go to www.parallax.com (shown in Figure 1-4).
√ Point at the Downloads menu to display the options.
√ Point at the BASIC Stamp Software link and click to select it.

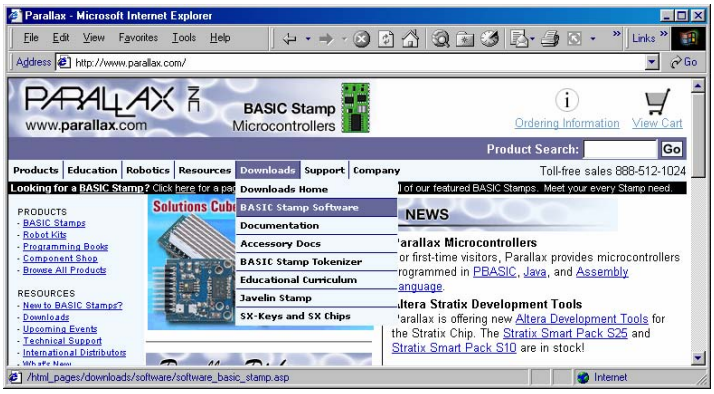


Figure 1-4
The Parallax Web Site:

www.parallax.com

√ When you get to the BASIC Stamp Software page, find a BASIC Stamp Windows Editor download with a version number of 2.0 or higher.

√ Click the Download icon. In Figure 1-5, the Download icon looks like a file folder to the right of the description: "BASIC Stamp Windows Editor Version 2.0 Beta 1 (6MB)".

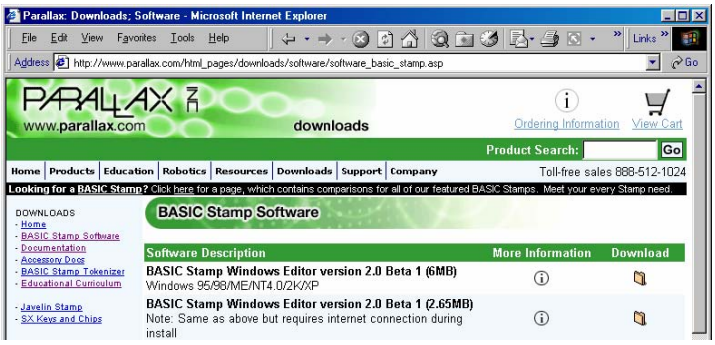


Figure 1-5
The Parallax Web Site Downloads Page

√ When the File Download window shown in Figure 1-6 appears, select: Save this program to disk.

√ Click the OK button.

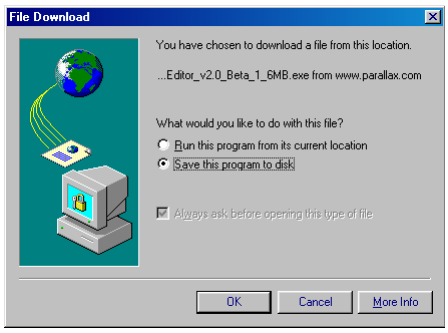


Figure 1-6
File Download Window

Figure 1-7 shows the Save As window that appears next. You can use the Save in field to browse your computer's hard drives to find a convenient place to save the file.

√ After choosing where to save the file you are downloading, click the Save button.

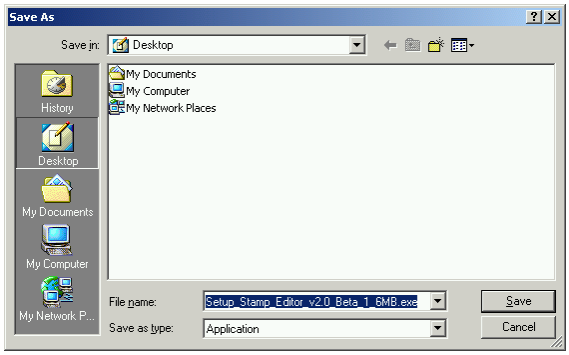


Figure 1-7
Save As Window

Selecting a place to save the file

√ Wait while the BASIC Stamp Editor installation program downloads (shown in Figure 1-8). This may take a while if you are using a modem connection.

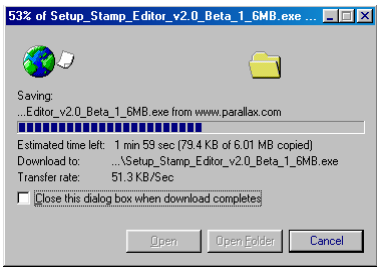


Figure 1-8
Download
Progress Window

√ When the download is complete, leave the window shown in Figure 1-9 open while you skip to the next section - Activity #2: Installing the Software.

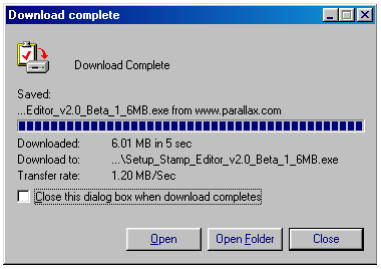


Figure 1-9
Download
Complete
Window

Go to Activity #2: Installing the Software.

Other free downloads at the Parallax web site include:

- This text and other Stamps in Class texts
- Robot videos
- More free software
- Hundreds of applications and experiments you can try!

Finding the Software on the Parallax CD

You can also install the BASIC Stamp Editor from the Parallax CD, but the CD has to be newer than April, 2003 so that you can get the version of the BASIC Stamp Editor that is compatible with the examples in this text. You can find the Parallax CD's Year and Month by examining the labeling on the front of the CD.

√ Place the Parallax CD into your computer's CD drive. The Parallax CD browser is called the Welcome application. It's shown in Figure 1-10 and it should run as soon as you load the CD into your computer's CD drive.

√ If the Welcome application does not automatically run, double-click My Computer, then double-click your CD drive, and then double-click Welcome.
√ Click the Software link shown in Figure 1-10.

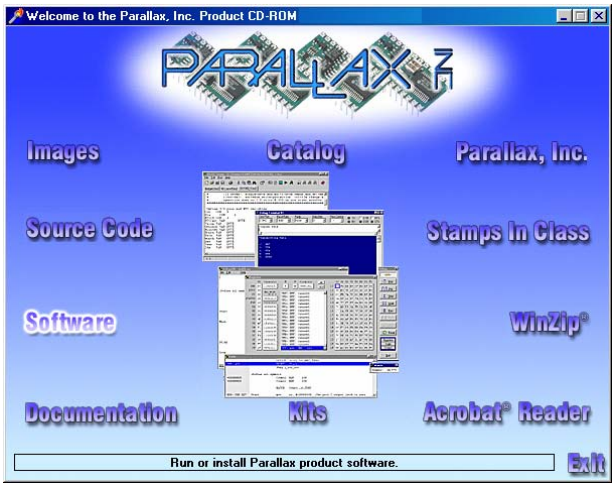


Figure 1-10
The Parallax CD Browser

√ Click the + next to the BASIC Stamps folder shown in Figure 1-11.
√ Click the + next to the Windows folder.
√ Click the floppy diskette icon labeled "Stamp 2/2e/2sx/2p/2pe (stampw.exe)".
√ Continue through Activity #2: Installing the Software.

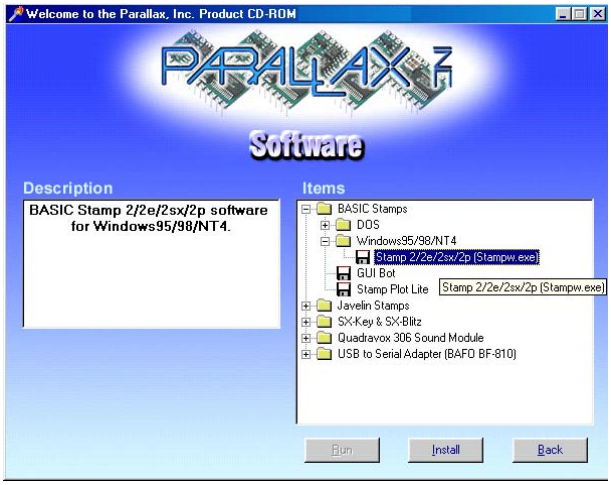


Figure 1-11
The Parallax CD Browser

Select the BASIC Stamp Editor installation program from the Software page.

Free downloads at the Parallax web site are included in the Parallax CD, but only up to the date the CD was created. The date on the front of the CD indicates when it was created. If the CD is just a month or two old, you will probably have the most up-to-date material. If it's an older CD, consider requesting a new one from Parallax or downloading the files you need from the Parallax web site.

ACTIVITY #2: INSTALLING THE SOFTWARE

By now, you have either downloaded the BASIC Stamp Editor Installer from the Parallax web site or located it on the Parallax CD. Now it's time to run the BASIC Stamp Editor Installer.

Installing the Software Step by Step

√ If you downloaded the BASIC Stamp Editor Installer from the Internet, click the Open button on the Download Complete window shown in Figure 1-12.

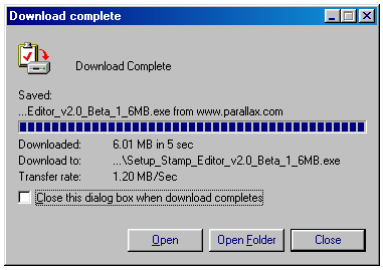


Figure 1-12
Download Complete Window

If you skipped here from the "Downloading the Software from the Internet" section, click the Open button now.

√ If you located the software on the Parallax CD, click the Install button shown in Figure 1-13.

Figure 1-13
The Parallax CD Browser

Install button located near bottom of window.

√ When the BASIC Stamp Editor…InstallShield Wizard window opens, click the Next button shown in Figure 1-14.

Figure 1-14
InstallShield Wizard for the BASIC Stamp Editor

√ Select Typical for your setup type as shown in Figure 1-15.
√ Click the Next button.

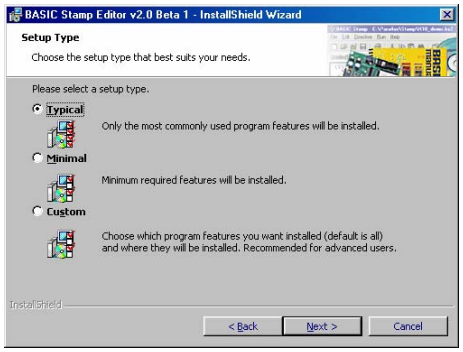


Figure 1-15
Setup Type

√ When the InstallShield Wizard tells you it is "Ready to Install the Program", click the Install button shown in Figure 1-16.

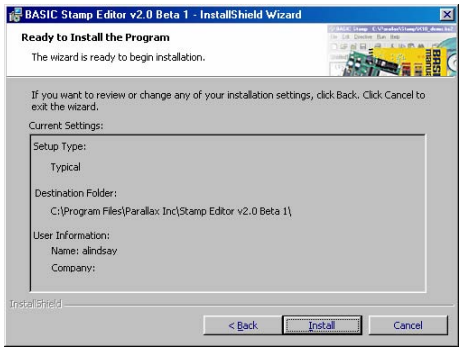


Figure 1-16
Ready to Install

Click the Install button.

√ When the InstallShield Wizard window tells you "InstallShield Wizard Completed" as shown in Figure 1-17, click Finish.

Congratulations! Your BASIC Stamp Editor is now installed.

Figure 1-17
InstallShield Wizard Completed

ACTIVITY #3: SETTING UP THE HARDWARE AND TESTING THE SYSTEM

The BASIC Stamp needs to be connected to power for it to run. It also needs to be connected to a PC so it can be programmed. After making these connections, you can use the BASIC Stamp Editor to test the system. This activity will show you how.

Computer Serial Cable Setup

The Board of Education or BASIC Stamp HomeWork Board should be connected to your PC or laptop by either a serial cable or a USB to Serial Adapter.

√ If you are using a serial cable, connect it to an available COM port on the back of your computer as shown in Figure 1-18.

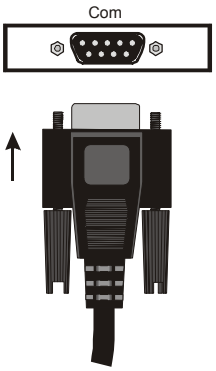


Figure 1-18
PC or Laptop COM Port

Plug the serial cable into an available COM port on your PC or laptop.

√ If you are using a USB to Serial Adapter, follow the hardware and software installation instructions that are supplied with the product.

FTDI's US232B/LC USB to Serial Adapter:

At the time of this writing, the US232B/LC USB to Serial Adapter made by Future Technology Devices International is the recommended adapter for use with Parallax products. The US232B/LC comes with the hardware shown in Figure 1-19 and a mini-CD ROM with drivers for use with various operating systems including Microsoft Windows®.

US232B/LC Driver Software Downloads: The software drivers and other information about this product can be downloaded from: http://www.ftdichip.com/FT232.htm.

Figure 1-19

FTDI's US232B/LC USB to Serial Adapter

This adapter is Parallax Stock# 800-00030. It comes with a software CD (not shown).

Now that your programming cable is connected to your computer, it's time to assemble the hardware.

√ If you have a BASIC Stamp and Board of Education, follow the instructions in the next section, Board of Education Connection Instructions.

√ If you have a BASIC Stamp HomeWork board, skip to the BASIC Stamp HomeWork Board Connection Instructions on page 18.

√ If your equipment is already hooked up, skip to the Testing for Communication section on page 21.

Board of Education Connection Instructions

If you have a BASIC Stamp and Board of Education, Figure 1-20 shows the hardware you will need to get started.

Required Hardware

(1) Strip of four rubber feet
(1) Battery pack
(1) BASIC Stamp 2
(1) Board of Education
(4) New AA alkaline batteries (not included)

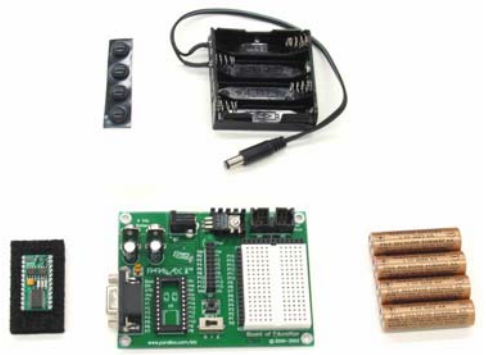


Figure 1-20
Getting Started Hardware for the BASIC Stamp and Board of Education

Connecting the Hardware

The rubber feet are shown in Figure 1-21, and they should be affixed to the underside of your Board of Education. The Board of Education has circles on its underside that show where each rubber foot should be attached.

√ Remove each rubber foot from the adhesive strip and affix it to the underside of the Board of Education.

Figure 1-21
Rubber Feet (left) Affixed to Underside of the Board of Education (right)

The Board of Education Rev C has a 3-position switch (see Figure 1-22). Position-0 is for turning the power to the Board of Education completely off. Regardless of whether or not you have a battery or power supply connected to the Board of Education Rev C, when the 3-posiiton switch is set to 0, the device is off.

√ Set the 3-position switch on the Board of Education to position-0.

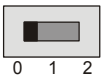


Figure 1-22
3-position Switch

Set to position-0 to turn off the power.

Only the Board of Education Rev C has a 3-position switch.

If you have a Board of Education Rev A or B:

- When directed to set the 3-position switch to position-0, turn off power by disconnecting the battery pack (the reverse of Figure 1-24, step 3).
- When directed to set the 3-position switch to either position-1 or position-2, plug the battery pack in as shown in Figure 1-24, step 3.

√ Load the batteries into the battery pack as shown in Figure 1-23. Make sure to follow the polarity (+ and -) markings on the inside of the battery pack's plastic case when inserting each battery.

Figure 1-23
Battery Pack

Polarity indicators on molded plastic (left) and loaded with correct polarity (right).

√ If your BASIC Stamp is not already plugged into your Board of Education, insert it into the socket shown in Figure 1-24, step-1.

Make sure your BASIC Stamp is right-side-up (as shown in Figure 1-24) before you insert it into the socket! If the BASIC Stamp is plugged into the socket upside-down, it could be damaged when you plug in power.

√ Make sure the pins are lined up properly with the holes in the socket, then press down firmly to seat it. The module should sink in about $^{1}/_{8}$ inch (3 mm).

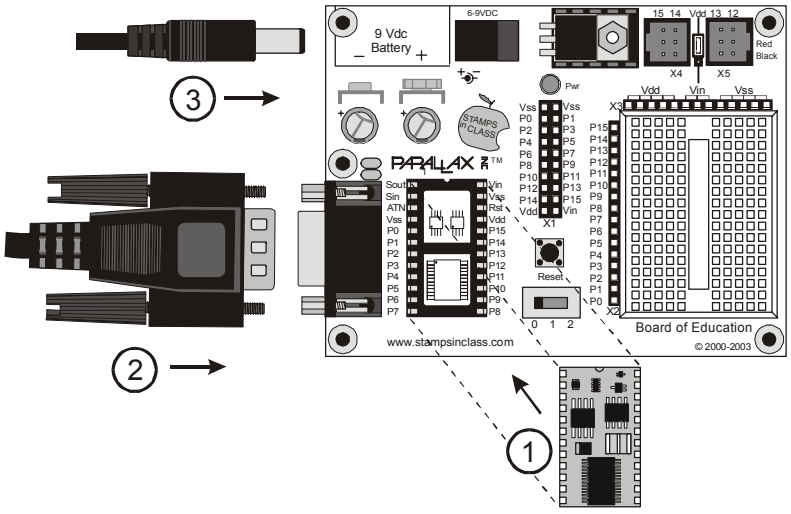


Figure 1-24
Board of Education, BASIC Stamp, Battery, and Serial Cable

Connect components in the order shown in the diagram.

√ Plug the serial cable into the Board of Education as shown in Figure 1-24, step-2.
√ Plug the battery pack into the 6-9 VDC battery jack as shown in Figure 1-24, step-3.
√ Move the 3-position switch from position-0 to position-1 to turn the power on.

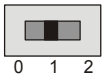


Figure 1-25
3-position Switch

Set to position-1 to turn the power back on.

√ The green light labeled Pwr on the Board of Education should now be on.

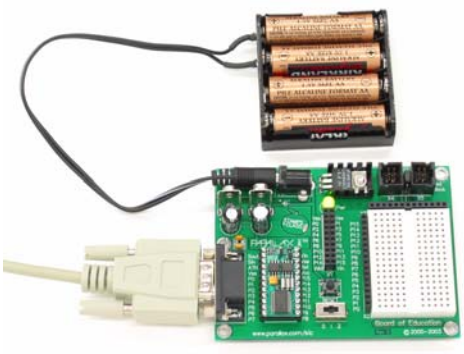


Figure 1-26
BASIC Stamp and Board of Education Connected and Ready to Program

√ Skip to the Testing for Communication section on page 21.

BASIC Stamp HomeWork Board Connection Instructions

This section will guide you through connecting your BASIC Stamp to your computer and a (battery) power supply if you have a BASIC Stamp HomeWork Board.

Required Hardware

√ Collect the following parts from your kit, shown in Figure 1-27
(1) Basic Stamp HomeWork Board
(1) Strip of four rubber feet
(1) New 9 V battery (not included)

(1) BASIC Stamp HomeWork Board

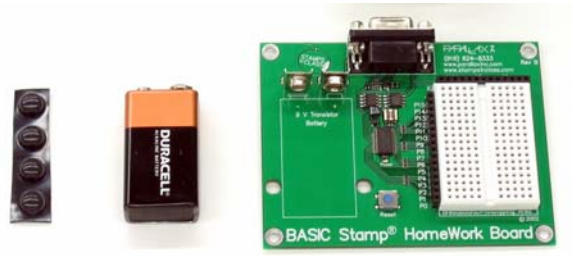


Figure 1-27
Getting Started Hardware for the BASIC Stamp HomeWork Board

√ Remove each rubber foot from its adhesive strip and affix it to the underside of the HomeWork Board next to each plated hole at each corner of the board as shown in Figure 1-28, making sure not to cover up the holes.

Figure 1-28
Rubber Feet

√ Connect the serial cable and battery to the HomeWork Board (Figure 1-29, steps 1 and 2).

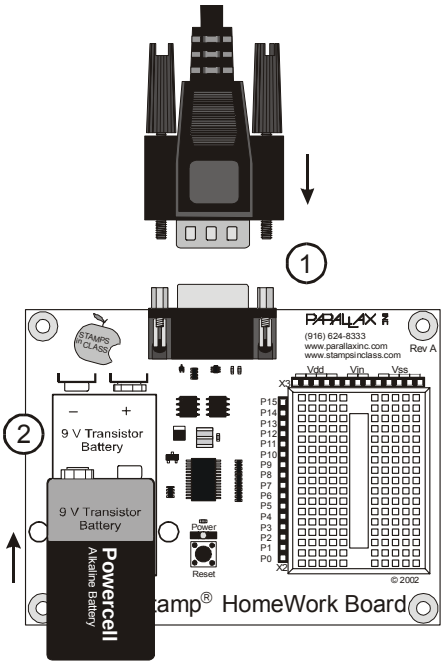


Figure 1-29
HomeWork Board and Serial Cable

Plug the serial cable and 9 V battery into the HomeWork Board.

Figure 1-30 shows the BASIC Stamp HomeWork Board connected to its battery power supply and serial programming cable.

The green Pwr light does not come on when you connect the battery. It will light up only when you have a program running.

You are now ready to test the programming connection between the BASIC Stamp and your PC/laptop.

Figure 1-30
BASIC Stamp HomeWork Board Ready to Program

Testing for Communication

√ First, run your BASIC Stamp Editor by double-clicking the shortcut on your desktop. It should look similar to the one shown in Figure 1-31.

Figure 1-31
BASIC Stamp Editor Shortcut

Look for a shortcut similar to this one on your computer's desktop.

The Windows Start Menu can also be used to run the BASIC Stamp Editor. Click your Windows *Start Button*, then select *Programs → Parallax, Inc. → Stamp Editor 2…*, then click the *BASIC Stamp Editor* Icon.

Your BASIC Stamp Editor window should look similar to the one shown in Figure 1-32.

The first time you run your BASIC Stamp Editor, it may display some messages and a list of your COM ports found by the software.

- √ If you know the number of the COM port your BASIC Stamp is connected to, check to make sure it is included in the list.
- √ If it is not included in the list, follow the BASIC Stamp Editor's instructions for adding a COM port.
- √ If you're not sure about your COM port, click OK for now.

√ To make sure your BASIC Stamp is communicating with your computer, click the Run menu, then select Identify.

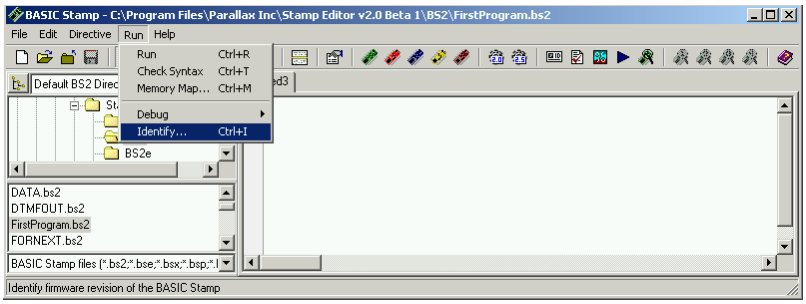


Figure 1-32
BASIC Stamp Editor

An Identification window similar to the one shown in Figure 1-33 will appear. The example in the figure shows that a BASIC Stamp 2 has been detected on COM2.

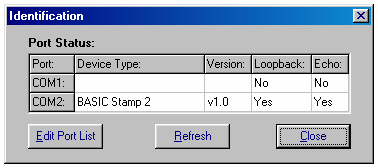


Figure 1-33
Identification Window

Example: BASIC Stamp 2 found on COM2.

√ Check the Identification window to make sure a BASIC Stamp 2 has been detected on one of the COM ports. If the BASIC Stamp 2 has been detected, then you are ready for Activity #4: Your First Program.

√ If the Identification window does not detect a BASIC Stamp 2 on any of the COM ports, go to page 301 (Appendix A: PC to BASIC Stamp Communication Trouble-Shooting).

ACTIVITY #4: YOUR FIRST PROGRAM

The first program you will write and test will tell the BASIC Stamp to send a message to your PC or laptop. Figure 1-34 shows how the BASIC Stamp sends a stream of ones and zeros to communicate the text characters displayed by the PC or laptop. These ones and zeros are called binary numbers. The BASIC Stamp Editor software has the ability to detect and display these messages as you will soon see.

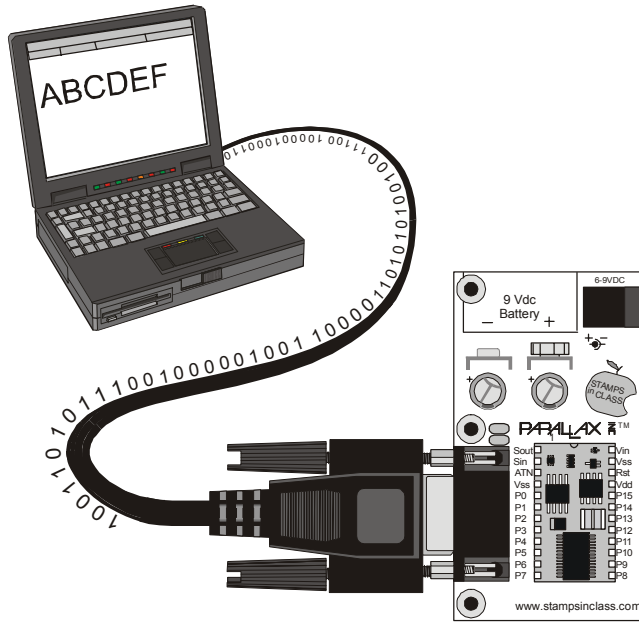


Figure 1-34
Messages from the BASIC Stamp to Your Computer

Your First Program

The example programs that you will type into the BASIC Stamp Editor and download to the BASIC Stamp will always be shown with a gray background. Here is an example:

Example Program: HelloBoeBot.bs2

```
' Robotics with the Boe-Bot - HelloBoeBot.bs2
' BASIC Stamp sends a text message to your PC/laptop.

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

DEBUG "Hello, this is a message from your Boe-Bot."

END
```

You will enter this program into the BASIC Stamp Editor. Some lines of the program are made automatically by clicking buttons on the toolbar. Other lines are made by typing them in from the keyboard.

√ Begin by clicking the BS2 icon (the green diagonal chip) on the toolbar, shown highlighted in Figure 1-35. If you hold your cursor over this button, its flyover help description "Stamp Mode: BS2" will appear.

√ Next, click on the gear icon labeled "2.5" shown highlighted in Figure 1-36. Its flyover help description is "PBASIC Language: 2.5".

Figure 1-35
BS2 Icon

Clicking on this button will automatically place `'{$STAMP BS2}` *at the beginning of your program.*

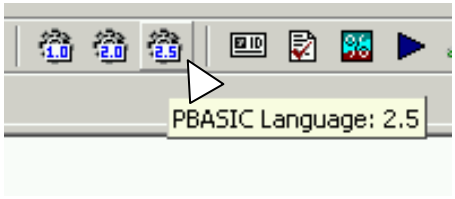


Figure 1-36
PBASIC 2.5 Icon

Clicking on this button will automatically place `'{$PBASIC 2.5}` *at the beginning of your program.*

ALWAYS use these toolbar buttons to add these two lines as the beginning of every program! Compiler directives use braces { }. If you try to type in these parts of your program, you may accidentally use parentheses ( ) or square brackets [ ]. If you do this, your program will not work.

√ Type the rest of the program into the BASIC Stamp Editor exactly as shown in Figure 1-37. Notice that the first two lines are above the compiler directives, and the rest of the program is below the compiler directives.

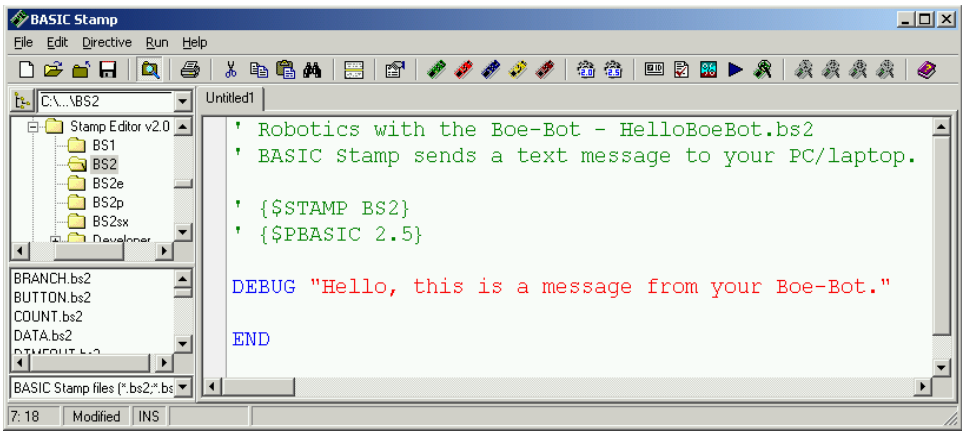


Figure 1-37 HelloBoeBot.bs2 Entered into the BASIC Stamp Editor

√ Save your work by clicking File and selecting Save, (shown in Figure 1-38).

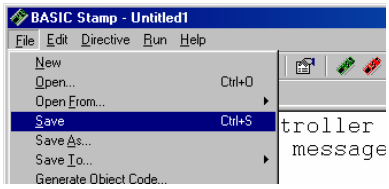


Figure 1-38
Saving the Program HelloBoeBot.bs2

√ Enter the name HelloBoeBot.bs2 into the File name field near the bottom of the Save As window as shown in Figure 1-39.

√ Click the Save button.

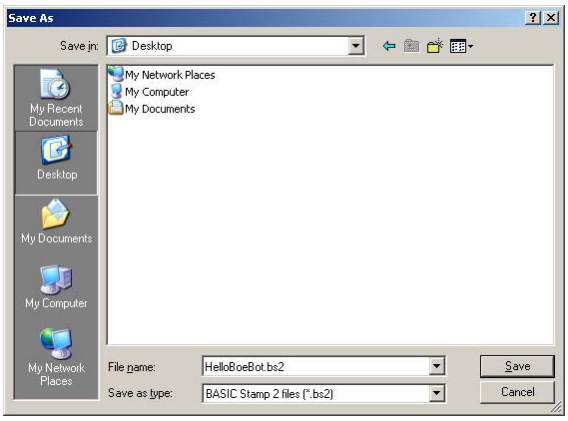


Figure 1-39
Entering the File Name

The next time you save, the BASIC Stamp Editor will automatically save to the same filename (HelloBoeBot.bs2) unless you tell it to save to a different filename by clicking *File* and selecting *Save As* (instead of just Save).

√ Click Run, and select Run from the menu that appears (by clicking it) as shown in Figure 1-40.

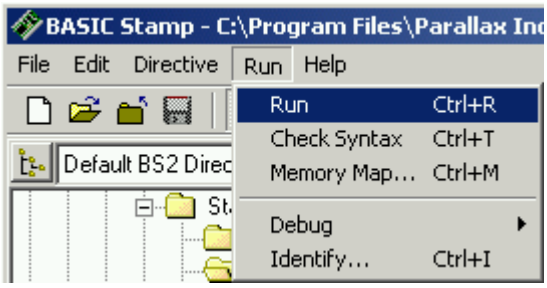


Figure 1-40
Running Your First Program HelloBoeBot.bs2

A Download Progress window will appear briefly as the program is transmitted from the PC or laptop to your BASIC Stamp. Figure 1-41 shows the Debug Terminal that should appear when the download is complete. You can prove to yourself that this is a message from the BASIC Stamp by pressing and releasing the Reset button on your Board of Education or HomeWork Board. Every time you press and release it, the program will re-run, and you will see another copy of the message displayed in the Debug Terminal.

√ Press and release the Reset button. Did you see a second "Hello…" message appear in the Debug Terminal?

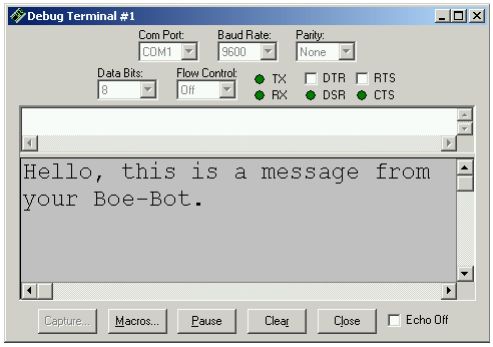


Figure 1-41
Debug Terminal

The Debug Terminal displays messages sent to the PC/laptop by the BASIC Stamp.

The BASIC Stamp Editor has shortcuts for most common tasks. For example, to run a program, you can press the 'Ctrl' and 'R' keys at the same time to run a program. You can also click the *Run* button. It's the blue triangle shown in Figure 1-42 that looks like a CD player's *Play* button. The flyover help (the Run hint) will appear if you point at the *Run* button with your mouse. You can get similar hints to find out what the other buttons do by pointing at them.

Figure 1-42
BASIC Stamp Editor
Shortcut Buttons

How HelloBoeBot.bs2 Works

The first two lines in the example program are called comments. A comment is a line of text that gets ignored by the BASIC Stamp Editor, because it's meant for a human reading the program, not for the BASIC Stamp. In PBASIC, everything to the right of an apostrophe is normally considered to be a comment by the BASIC Stamp Editor. The first comment tells which book the example program is from and what the program's filename is. The second comment contains a handy, one-line description that explains what the program does.

```
' Robotics with the Boe-Bot - HelloBoeBot.bs2
' BASIC Stamp sends a text message to your PC/laptop.
```

There are several special messages that you can send to the BASIC Stamp Editor by placing them inside comments (to the right of an apostrophe on a given line). These are called compiler directives, and every program in this text will use these two directives:

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

The first directive is called the Stamp directive, and it tells the BASIC Stamp Editor that you will be downloading the program to a BASIC Stamp 2. The second directive is called the PBASIC directive, and it tells the BASIC Stamp Editor that you are using version 2.5 of the PBASIC programming language.

A command is a word you can use to tell the BASIC Stamp to do a certain job. The first of the two commands in this program is called the **DEBUG** command:

```
DEBUG "Hello, this is a message from your Boe-Bot."
```

This is the command that tells the BASIC Stamp to send a message to the PC using the serial cable.

The second command is called the **END** command:

```
END
```

This command is handy because it puts the BASIC Stamp into low power mode when it's done running the program. In low power mode, the BASIC Stamp waits for either the Reset button to be pressed (and released), or for a new program to be loaded into it by the BASIC Stamp Editor. If the Reset button on your board is pressed, the BASIC Stamp will run the program you loaded into it again. If a new program is loaded into it, the old one is erased, and the new program begins to run.

Your Turn – DEBUG Formatters and Control Characters

A **DEBUG** formatter is a code-word you can use to make the message the BASIC Stamp sends look a certain way in the Debug Terminal. **DEC** is an example of a formatter that makes the Debug Terminal display a decimal value. An example of a control character is **CR**, which sends a carriage return to the Debug Terminal. The text or numbers that come after a **CR** will appear on the line below characters that came before it. You can modify your program so that it contains more **DEBUG** commands along with some formatters and control characters. Here's an example of how to do it:

√ First, save the program under a new name by clicking File and selecting Save As.

√ A good new name for the file would be HelloBoeBotYourTurn.bs2.
√ Modify the comments at the beginning of the program so that they read:

```
' Robotics with the Boe-Bot - HelloBoeBotYourTurn.bs2
' BASIC Stamp does simple math, and sends the results
' to the Debug Terminal.
```

√ Add these three lines between the first **DEBUG** command and the **END** command:

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

√ Save the changes you made by clicking File and selecting Save.

Your program should now look like the one shown in Figure 1-43.

Run your modified program. Hint: you will have to either click Run from the Run menu again, like in Figure 1-40, or click the Run button, like in Figure 1-42.

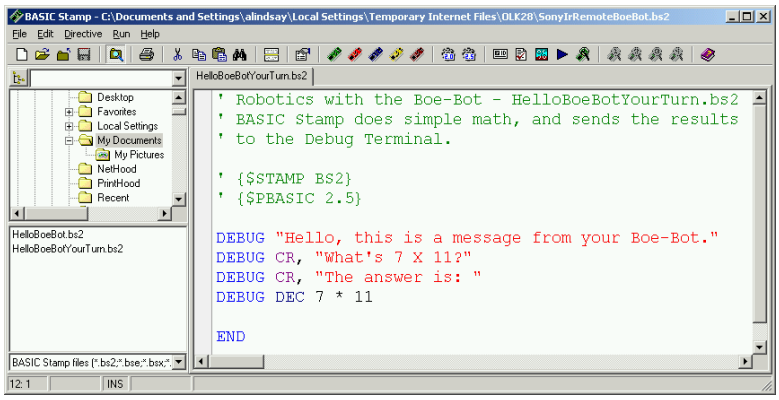


Figure 1-43
Modified HelloBoeBot.bs2

Check your work against the example program shown here.

Your Debug Terminal should now resemble Figure 1-44.

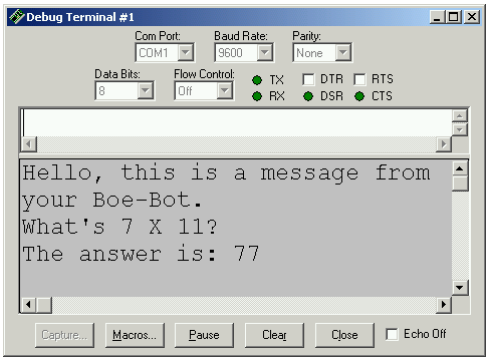


Figure 1-44
Modified HelloBoeBot.bs2 Debug Terminal Output

Make sure that when you re-run your program, you get the results you expect.

Where did my Debug Terminal go? Sometimes the Debug Terminal gets hidden behind the BASIC Stamp Editor window. You can bring it back to the front by using the *Run* menu as shown at the left of Figure 1-45, the *Debug Terminal 1* shortcut button shown at the right of the figure, or the F12 key on your keyboard.

Figure 1-45

Debug Terminal 1 to Foreground

Using the menu (left) and using the shortcut button (right).

ACTIVITY #5: LOOKING UP ANSWERS

The example program you just finished introduced two PBASIC commands: **`DEBUG`** and **`END`**. You can find out more about these commands and how they are used by looking them up, either in the BASIC Stamp Editor's Help or in the *BASIC Stamp Manual*. This activity guides you through an example of looking up **`DEBUG`** using the BASIC Stamp Editor's Help and the *BASIC Stamp Manual*.

Using the BASIC Stamp Editor's Help

√ In the BASIC Stamp Editor, Click Help, then select Index as shown in Figure 1-46.

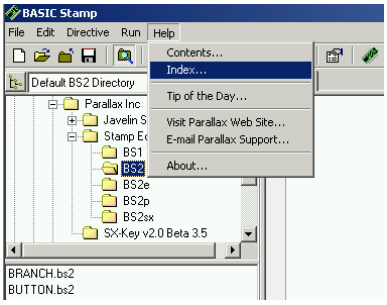


Figure 1-46
Selecting Index from the Help Menu

√ Type **DEBUG** into the field labeled Type in the keyword to find: (shown in Figure 1-47).

√ When the word **DEBUG** appears in the list below the field you are typing in, double-click it, then click the Display button.

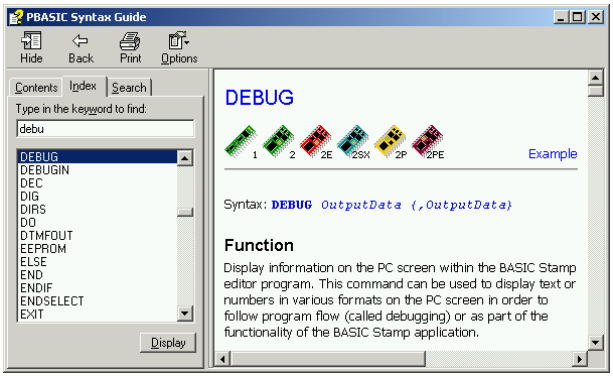


Figure 1-47
Looking up the DEBUG Command Using Help

Your Turn

√ Use the scrollbar to review the **DEBUG** command article. Notice that it has lots of explanations and example programs you can try.

√ Click the Contents tab, and find **DEBUG** there.

√ Click the Search tab, and run a search for the word **DEBUG**.
√ Repeat this process for the **END** command.

Getting and Using the BASIC Stamp Manual

The *BASIC Stamp Manual* is available for free download from the Parallax web site, and it's also included on the Parallax CD. It can also be purchased as a bound and printed manual.

Downloading the *BASIC Stamp Manual* from the Parallax Web Site

√ Using a web browser, go to www.parallax.com.
√ Point at the Downloads menu to display the options.
√ Point at the Documentation link and click to select it.
√ When you get to the BASIC Stamp Documentation page, find the *BASIC Stamp Manual.*
√ Click the Download icon that looks like a file folder to the right of the description: "BASIC Stamp Manual Version 2.0 (3.2 MB)".

Viewing the *BASIC Stamp Manual* on the Parallax CD

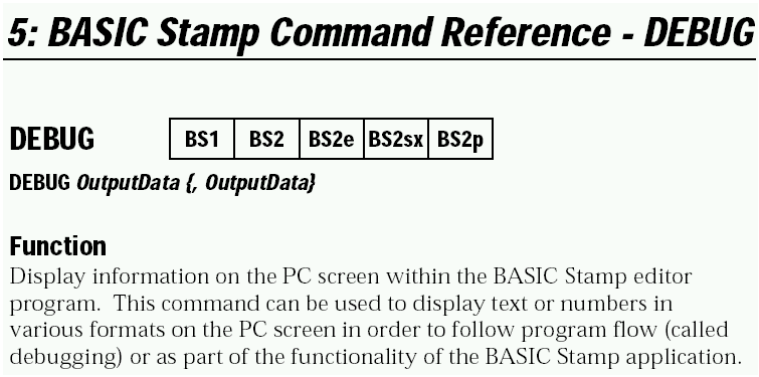
√ Click the Documentation link.
√ Click the + next to the BASIC Stamps folder.
√ Click the *BASIC Stamp Manual* book icon.
√ Click the View button.

√ Figure 1-48 shows an excerpt from the *BASIC Stamp Manual* Contents section (Page 2). It shows that information on the **DEBUG** command can be found on page 97.

Figure 1-48 Finding the DEBUG Command in the Table of Contents

Figure 1-49 shows an excerpt from the *BASIC Stamp Manual.* The **DEBUG** command is explained in detail here.

√ Briefly look over the *BASIC Stamp Manual* explanation of the **DEBUG** command.
√ Count the number of example programs in the **DEBUG** section. How many are there?



5: BASIC Stamp Command Reference - DEBUG

DEBUG BS1 | BS2 | BS2e | BS2sx | BS2p

DEBUG *OutputData {, OutputData}*

Function

Display information on the PC screen within the BASIC Stamp editor program. This command can be used to display text or numbers in various formats on the PC screen in order to follow program flow (called debugging) or as part of the functionality of the BASIC Stamp application.

Figure 1-49 Reviewing the **DEBUG** Command in the *BASIC Stamp Manual*

Your Turn

√ Use the *BASIC Stamp Manual* index to look up the **DEBUG** command.
√ Look up the **END** command in the *BASIC Stamp Manual*.

ACTIVITY #6: INTRODUCING ASCII CODE

In Activity #4: Your First Program, you used the **DEC** formatter with the **DEBUG** command to display a decimal number in the Debug Terminal. But what happens if you don't use the **DEC** formatter with a number? If you use the **DEBUG** command followed by a number with no formatter, the BASIC Stamp will read that number as an ASCII code.

Programming with ASCII Code

ASCII is short for American Standard Code for Information Interchange. Most microcontrollers and PC computers use this code to assign a number to each keyboard function. Some numbers correspond to keyboard actions, such as cursor up, cursor down, space, and delete. Other numbers correspond to printed characters and symbols. The numbers 32 through 126 correspond to those characters and symbols that the BASIC Stamp can display in the Debug Terminal. The following program will use ACSII code to display the words "BASIC Stamp 2" in the Debug Terminal.

Example Program – AsciiName.bs2

√ Enter and run AsciiName.bs2.

Remember to use the toolbar icons to place Compiler Directives into your programs!

'{$STAMP BS2} - Use the diagonal green electronic chip icon.
'{$PBASIC 2.5} - Use the gear icon labeled 2.5.

You can see a picture of these icons again on page 24.

```
' Robotics with the Boe-Bot - AsciiName.bs2
' Use ASCII code in a DEBUG command to display the words BASIC Stamp 2.

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

DEBUG 66,65,83,73,67,32,83,116,97,109,112,32,50

END
```

How AsciiName.bs2 Works

Each letter in the **DEBUG** command corresponds to one ASCII code symbol that appeared in the Debug Terminal.

```
DEBUG 66,65,83,73,67,32,83,116,97,109,112,32,50
```

66 is the ASCII code for capital "B", 65 is the code for capital "A" and so on. 32 is the code for a space between characters. Notice that each code number was separated with a comma. The commas allow the one instance of **DEBUG** to execute each symbol as a separate command. This is much easier to type than 12 separate **DEBUG** commands.

Your Turn – Exploring ASCII Code

√ Save AsciiName.bs2 as AsciiRandom.bs2
√ Pick 12 random numbers between 32 and 127.
√ Replace the ASCII code numbers in the program with the numbers you chose.
√ Run your modified program to see what you get!

The *BASIC Stamp Manual* Appendix A has a chart of ASCII code numbers and their corresponding symbols. You can look up the corresponding code numbers to spell your own name.

√ Save AsciiRandom.bs2 as YourAsciiName.bs2
√ Look up the ASCII Chart in the *BASIC Stamp Manual*.
√ Modify the program to spell your own name.
√ Run the program to see if you spelled your name correctly.
√ If you did, good job, and save your program!

ACTIVITY #7: WHEN YOU'RE DONE

It's important to disconnect the power from your BASIC Stamp and Board of Education or HomeWork Board for several reasons. First, your batteries will last longer if the system is not drawing power when you're not using it. Second, in future experiments, you will build circuits on the Board of Education's or HomeWork Board's prototyping area.

Circuit prototypes should never be left unattended with a battery or power supply connected. You never know what kind of accident might occur when you are not there.

Always disconnect the power from your Board of Education or HomeWork Board, even if you only plan on leaving it alone for a minute or two.

If you are in a classroom, your instructor may have extra instructions, such as disconnecting the serial cable, storing your Board of Education/HomeWork Board in a safe place, etc. Aside from those details, the most important step that you should always follow is disconnecting power when you're done.

Disconnecting Power

With the Board of Education Rev C, disconnecting power is easy:

√ If you are using the Board of Education Rev C, move the 3-position switch to position-0 by pushing it to the left as shown in Figure 1-50.

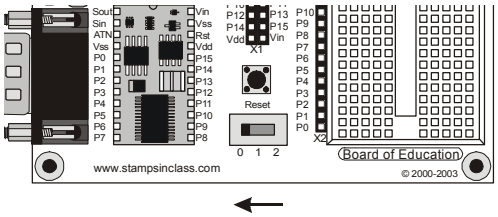


Figure 1-50
Switching Off the Power for the Board of Education Rev C

Do not remove the BASIC Stamp from its socket in the Board of Education!

Resist any temptation to store your Board of Education and BASIC Stamp separately. Every time the BASIC Stamp is removed and re-inserted into the socket on the Board of Education, mistakes may occur that can damage it. Although the BASIC Stamp is sometimes moved from one socket to another during a larger project, it will not be necessary during any of the activities in this text.

Disconnecting the BASIC Stamp HomeWork Board's power is easy too:

√ If you are using the BASIC Stamp HomeWork Board, disconnect the battery as shown in Figure 1-51.

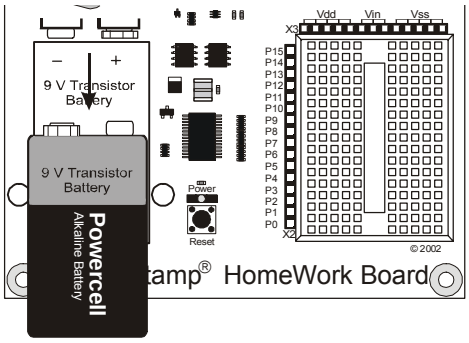


Figure 1-51
Disconnecting the Power for the HomeWork Board

The Board of Education Rev A or B also has to have its power disconnected, either by removing the battery or by unplugging the DC supply from the jack.

Your Turn

√ Disconnect power now.

SUMMARY

This chapter guided you through the following:

- An introduction to the BASIC Stamp
- 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
- An introduction to the BASIC Stamp, 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
- Using the **CR** control character and **DEC** formatter
- Using ASCII codes to transmit characters
- How to use the BASIC Stamp Editor's Help and the *BASIC Stamp Manual*
- 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 serial 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 you can do with each PBASIC command you learned in this chapter.
2. Explain what the asterisk does in 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. Write a program that uses a **DEBUG** instruction to display the solution to the math problem: 1 + 2 + 3 + 4.
2. Predict what you would expect to see if you removed the **DEC** formatter from this command. Use a PBASIC program to test your prediction.

   ```
   DEBUG DEC 7 * 11
   ```

3. Which lines can you delete in HelloBoeBotYourTurn.bs2 if you place the command shown below on the line just before the **END** command in the program? Test your hypothesis (your prediction of what will happen). Make sure to save HelloBoeBotYourTurn.bs2 with a new name to help keep track, like HelloBoeBotCh01Project03.bs2. Then make your modification, save and run your program.

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

Solutions

Q1. The BASIC Stamp 2 microcontroller module.
Q2. Binary numbers.
Q3. The Debug Terminal.
Q4. **DEBUG** and **END**.

E1. **DEBUG** – This command is used to send a message from the BASIC Stamp to the PC. The information is displayed on the Debug Terminal.
END – This command is used to terminate a PBASIC program and put the BASIC Stamp module into low-power mode.
E2. The asterisk multiplies the two operands 7 and 11, resulting in a product of 77. The asterisk is the multiply operator.
E3. To fix the problem, add a carriage return, the **CR** control character.

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

P1. Here is a program to display a solution to the math problem:

```
' Robotics with the Boe-Bot - HelloBoeBotCh01Project01.bs2
' Adds together 4 numbers with DEBUG
'{$STAMP BS2}
'{$PBASIC 2.5}

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

END
```

P2. Prediction: It will print the character "M". This program tests this prediction:

```
' Robotics with the Boe-Bot - HelloBoeBotCh01Project02.bs2
' Prints ASCII 7 * 11

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

DEBUG  7 * 11

END
```

P3. The last three **DEBUG** lines can be deleted. An additional **CR** is needed after the "Hello" message.

```
' Robotics with the Boe-Bot - HelloBoeBotCh01Project03.bs2
' Send message to Debug Terminal and do some math.

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

DEBUG "Hello, this is a message from your Boe-Bot.", CR

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

END
```

The output from the Debug Terminal is:

```
Hello, this is a message from your Boe-Bot.
What's 7 X 11?
The answer is: 77
```

This output is the same as it was with the three lines. This is an example of using commas to output a lot of information, using only one **DEBUG** statement.

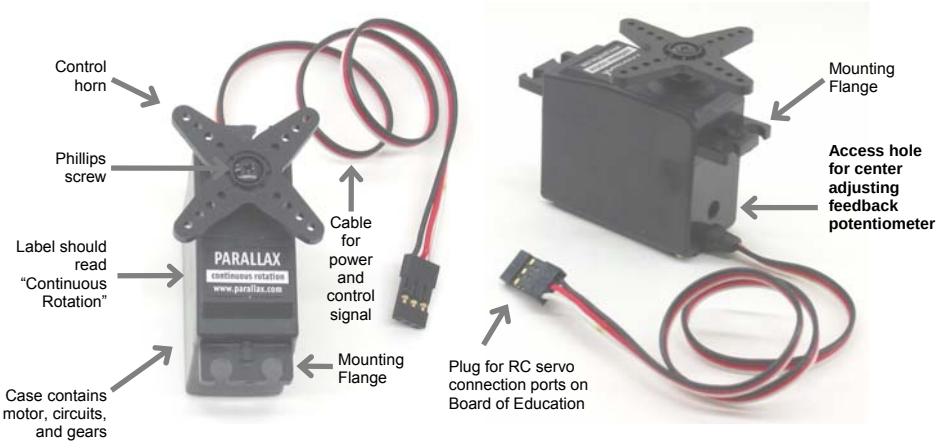
Chapter 2: Your Boe-Bot's Servo Motors

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



TIP: You may 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.

ACTIVITY #1: HOW TO TRACK TIME AND REPEAT ACTIONS

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 Rev C, 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, 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?* Excerpts –** This activity contains selected excerpts from the *What's a Microcontroller?* Student Guide v2.0.

√ 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?*

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.

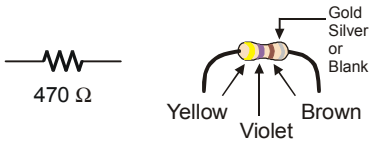


Figure 2-2
470 Ω Resistor Part Drawing

Schematic symbol (left) and part drawing (right)

The colored stripes indicate resistance values. See Appendix C: Resistor Color Codes 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.

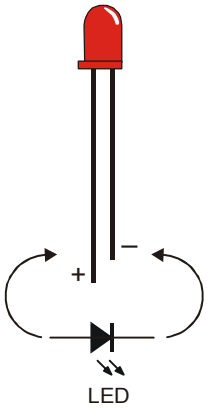


Figure 2-3
LED Part Drawing and Schematic Symbol

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

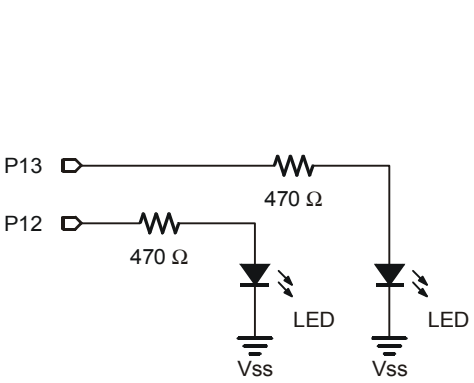
(2) LEDs – Red
(2) Resistors – 470 Ω (yellow-violet-brown)

Always disconnect power to your board before building or modifying circuits! For the Board of Education Rev C, 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.



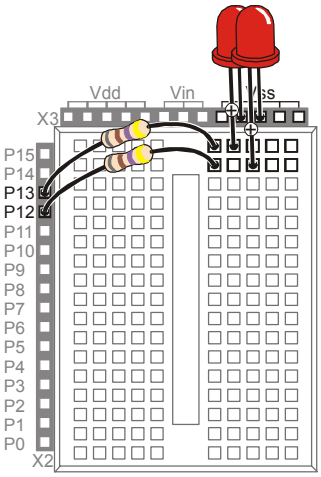


Figure 2-4 Two LEDs Connected to BASIC Stamp I/O Pins P13 and P12

Schematic (left) and wiring diagram (right).

What's an I/O pin? I/O stands for input/output. The BASIC Stamp 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 speakers, 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 D: Breadboarding Rules.

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.

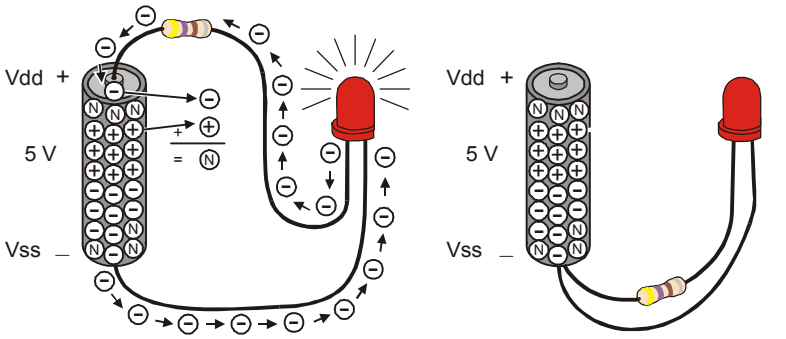


Figure 2-5
BASIC Stamp Switching

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

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.

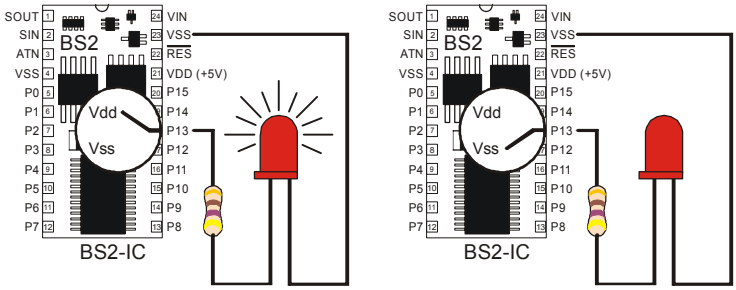


Figure 2-6
BASIC Stamp Switching

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`** loop. 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 serial 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.

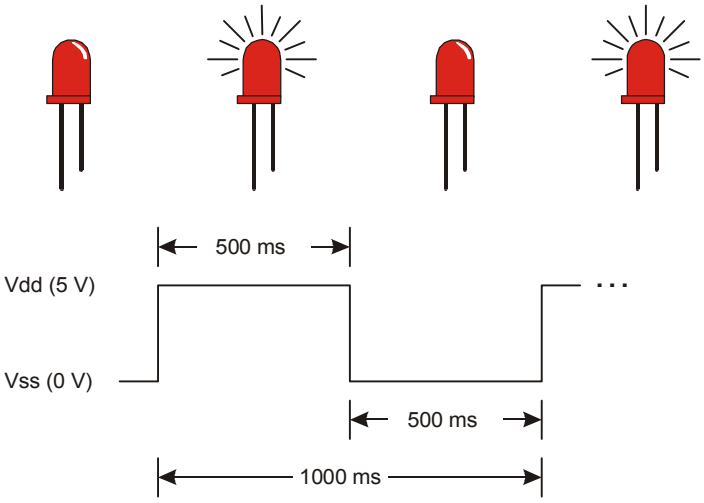


Figure 2-7
Timing Diagram for HighLowLed.bs2

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 ***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.

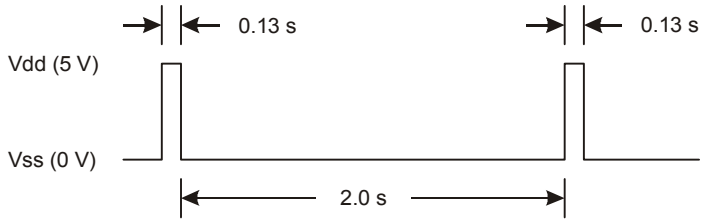


Figure 2-8
Timing Diagram for PulseP13Led.bs2

- √ 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.

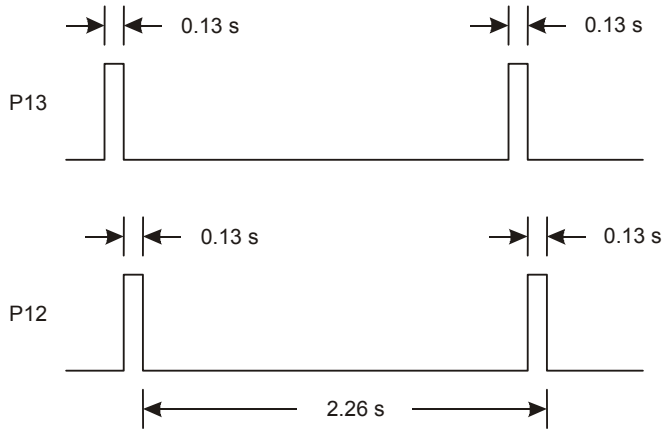


Figure 2-9
Timing Diagram for PulseBothLeds.bs2

The LEDs emit light for 0.13 seconds 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 the modified program 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 **PULSOUT** command that goes to P13.

```
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 are different because the amount of time the LED connected to P13 stays on is longer than the amount of time the LED connected to P12 stays on.

√ Try substituting 750 in the ***Duration*** argument for the **PULSOUT** command that goes to both LEDs.

```
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 previous activity will be used later to monitor the signals the BASIC Stamp sends to the servos to control their motion.

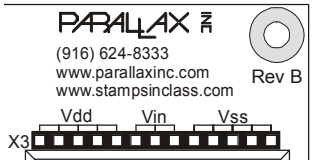
Parts for Connecting the Servos

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

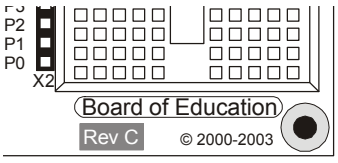
Finding the Connection Instructions for Your Carrier Board

There are three different Revs of the Board of Education and one Rev of the BASIC Stamp HomeWork Board. Boards of Education can either be Rev A, B, or C. The HomeWork Board is Rev B. Figure 2-10 shows examples of the labeling you might see on your board.

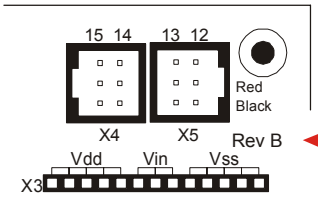
√ Examine the labeling on your carrier board and figure out whether you have a BASIC Stamp HomeWork Board Rev B or a Board of Education Rev C, B, or A.



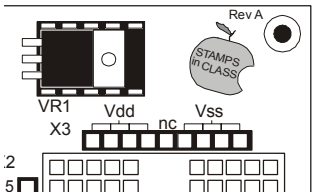
BASIC Stamp HomeWork Board Rev B



Board of Education Rev C



Board of Education Rev B



Board of Education Rev A

Figure 2-10
Examples of Rev Labels on the BASIC Stamp HomeWork Board and the Board of Education

√ Knowing the revision of your carrier board, skip to instructions (listed below) for connecting the servo to your board:

Page 60 → Board of Education Rev C
Page 63 → BASIC Stamp HomeWork Board

Board of Education Rev B

If you have a Board of Education Rev B, follow the instructions for the Board of Education Rev C throughout the text, always keeping these two points in mind:

- The Board of Education Rev B does not have a 3-positioin switch. You will have to disconnect battery pack's plug from the Board of Education's power jack when directed to set the 3-position switch to position-0. When directed to set the 3-position switch to position-1 or 2, you will have to plug the power in.
- The Board of Education Rev B does not have a jumper setting for power. Only use the 6 V battery pack as a power source for Board of Education Rev B Boe-Bot projects.

Board of Education Rev A

If you have a Board of Education Rev A, follow the instructions for the BASIC Stamp HomeWork Board throughout the text.

√ When you are done, go to Activity #4: Centering the Servos on page 66.

Connecting the Servos to the Board of Education Rev C

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

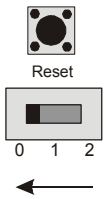


Figure 2-11
Disconnect Power

Figure 2-12 shows the servo header on the Board of Education Rev C. 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.

Use only alkaline AA (1.5 V) batteries. Avoid rechargeable batteries because they are 1.2 V instead of 1.5 V.

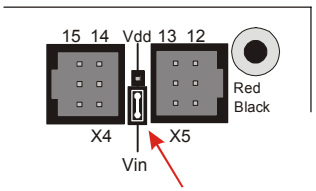
√ If you are using a 7.5 V, 1000 mA center positive DC supply, set the jumper to Vdd as shown on the right side of Figure 2-12.

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).

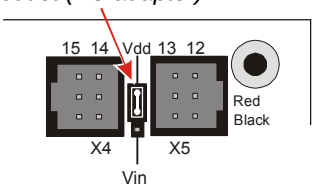


Figure 2-12
Selecting Your Servo's Power Supply on the Board of Education Rev C

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 Rev C. The jumper is set to Vin.

√ Connect your servos to your Board of Education Rev C as shown in Figure 2-13.

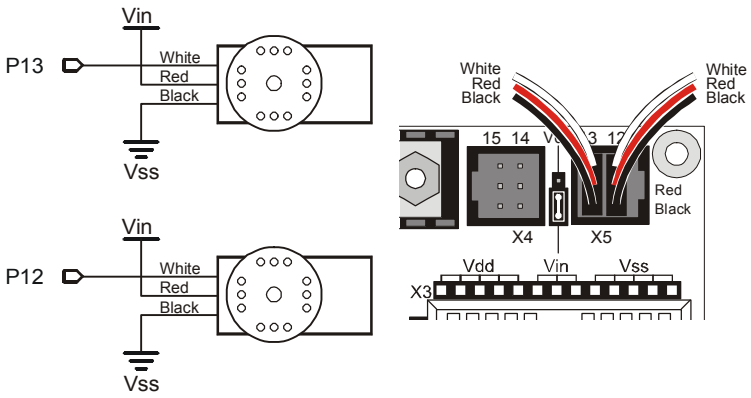


Figure 2-13
Servo Connection Schematic and Wiring Diagram for the Board of Education Rev C

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).

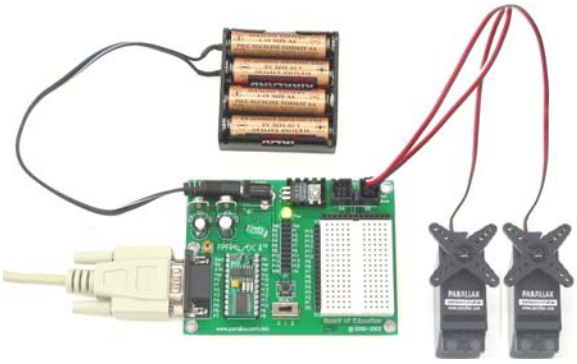


Figure 2-14
Board of Education with Servos and Battery Pack Connected

√ 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.

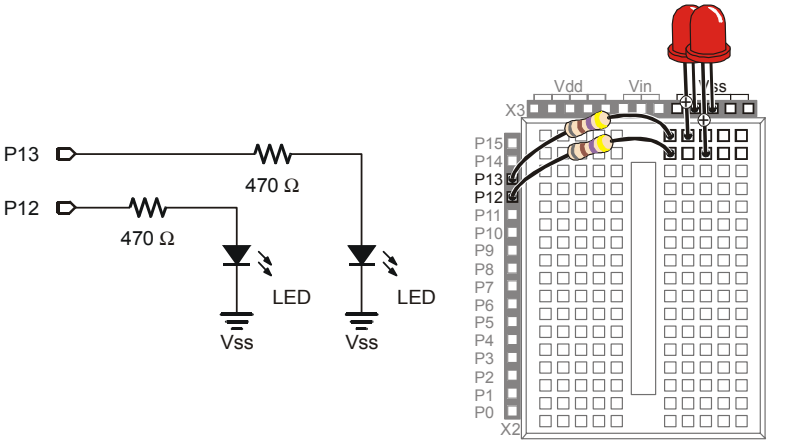


Figure 2-15
LED Servo Signal Monitor Circuit

Disconnecting Power – Special Instructions for the Board of Education Rev C

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

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

√ Move on to page 66 (Activity #4: Centering the Servos).

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
(2) Parallax Continuous Rotation Servos
(2) 3-pin male-male headers
(4) Jumper wires
(4) AA batteries – 1.5 V alkaline
(2) Built and tested LED circuits from the previous activity

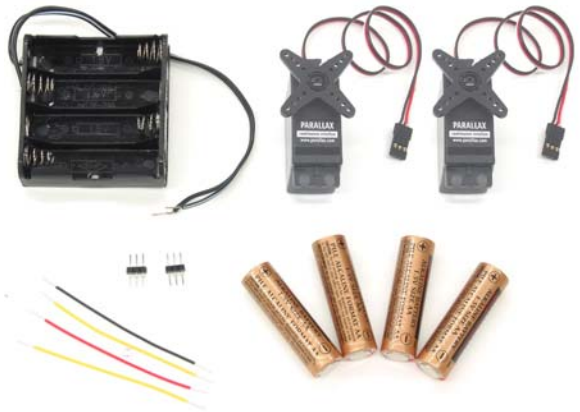


Figure 2-16
Servo Centering Parts for the HomeWork Board

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.

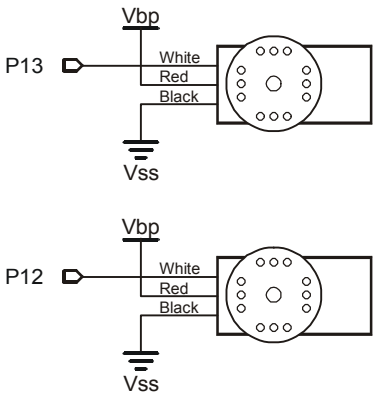


Figure 2-17
Servo Connection Schematic for the BASIC Stamp HomeWork Board.

- √ 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, and P12 all exactly match the wiring diagram.
- √ Connect the servo plugs to the male headers as shown on the right of Figure 2-18.
- √ 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 or Board of Education Rev A. Your BASIC Stamp is still powered by the 9 V battery.

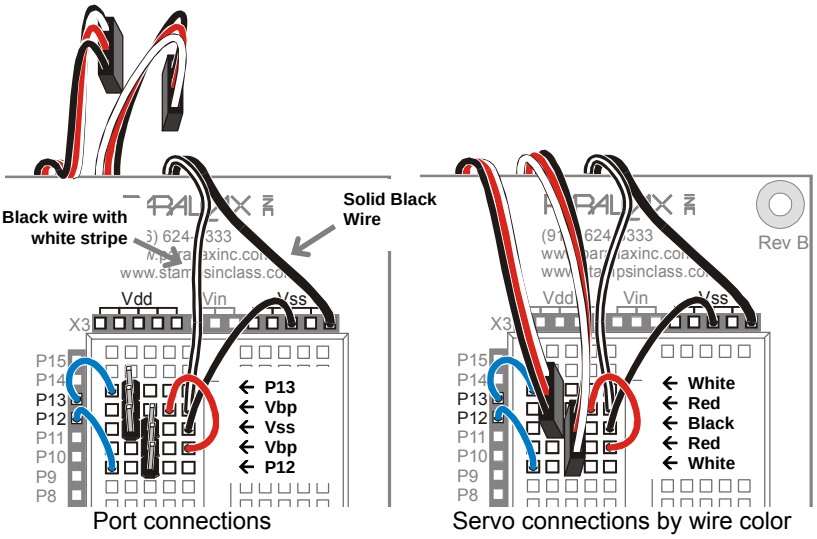


Figure 2-18 Servo Connection Wiring Diagram for the BASIC Stamp HomeWork Board

Left (build the servo ports).

Right (connect the servos).

Your setup will then resemble Figure 2-19.

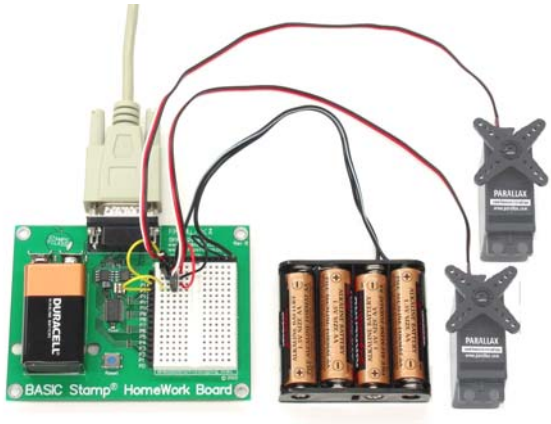


Figure 2-19 Dual Supplies and Servos Connected

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

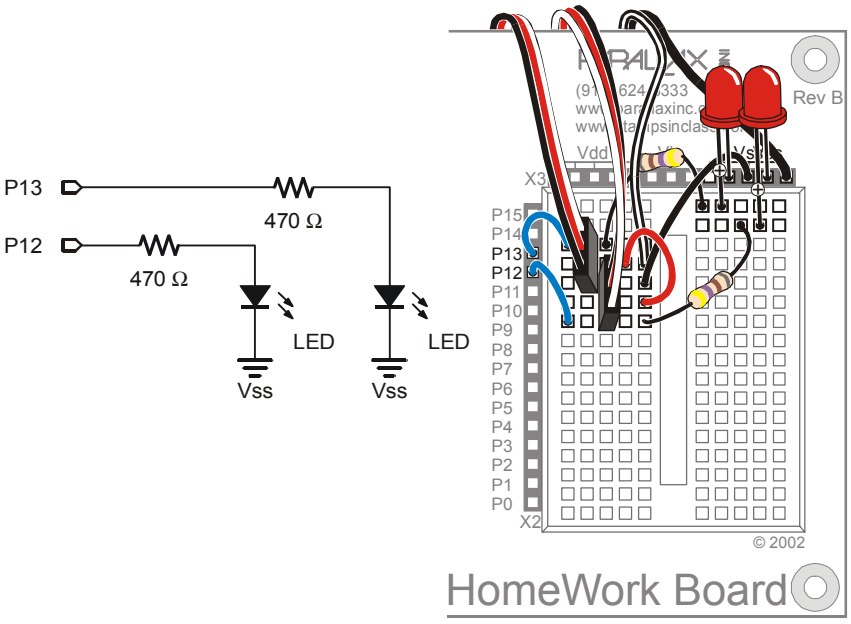


Figure 2-20
LED Servo Signal Monitor Circuit

√ 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 – Special Instructions 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.

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. Alternately, 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.

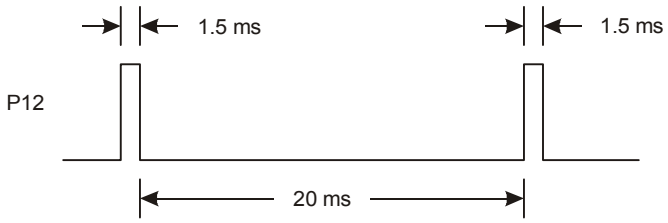


Figure 2-22
Timing Diagram for
CenterServoP12.bs2

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 servo connected to P12, you will repeat it with the servo connected to P13.

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

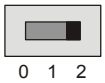


Figure 2-23
Set the 3-Position Switch to Position-2

√ 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.

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 when adjusting the servo.

Insert tip of Phillips screwdriver into potentiometer access hole.

Gently turn screwdriver to adjust potentiometer

Figure 2-24
Center Adjusting a Servo

√ 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, skip to the Your Turn section on page 70 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 Rev C.

√ 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.
√ 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 Rev C, set the 3-position switch to position-1. This disconnects power from the servo ports only. The BASIC Stamp, Vdd, Vin, and Vss will all still be connected to power.
- √ If you have a BASIC Stamp HomeWork Board, remove one battery from the battery pack, but leave the 9 V battery connected to the battery clip. This disconnects power from the servo ports, but power remains connected to the embedded BASIC Stamp, Vdd, Vin, and Vss.

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 either numbers or variables. 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.

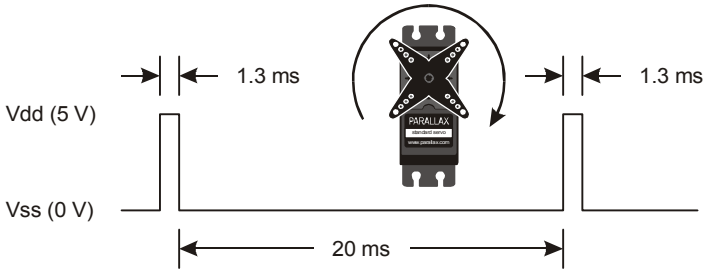


Figure 2-25
A 1.3 ms Pulse Train Turns the Servo Full Speed Clockwise

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.

You can use ServoP13Clockwise.bs2 to send this pulse train to the servo connected to P13.

Example Program: ServoP13Clockwise.bs2

Your entire system, including servos should be connected to power for this activity.

- √ If you have a Board of Education Rev C, set the 3-position switch to position-2. This connects power to the servo ports in addition to the position-1 power to the BASIC Stamp, Vdd, Vin, and Vss.
- √ If you have a BASIC Stamp HomeWork Board, replace the battery you removed from the battery pack. This will restore power to the servo ports. Also, connect the 9 V battery to the battery clip. This will supply power to the embedded BASIC Stamp, Vdd, Vin, and Vss.

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

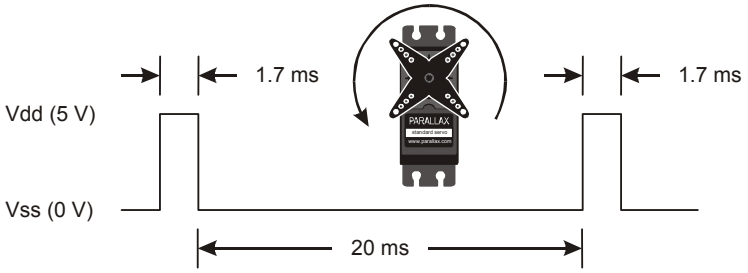


Figure 2-26
A 1.7 ms Pulse Train Makes the Servo Turn Full Speed Counterclockwise

√ 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
```

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 counterclockwise, 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 you 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 / 0.0246s = 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. That means setting the 3-posisiton switch to position-0 if you have a Board of Education Rev C. If you have a HomeWork Board, disconnect the 9 V battery from the battery clip and remove one battery from the battery pack.

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 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 are the command and argument that you can 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
```

Projects

1. Write a program that causes the 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. Millisecond is abbreviated "ms".

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 (nibble) will not work, since the maximum value a Nib 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 the LED on P14,
' to make it light dimly.
'{$STAMP BS2}
'{$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 79, 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

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 screwdriver $^{1}/_{8}$″ (3.18 mm) shaft)
(1) $^{1}/_{4}$″ Combination wrench (Optional)
(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.

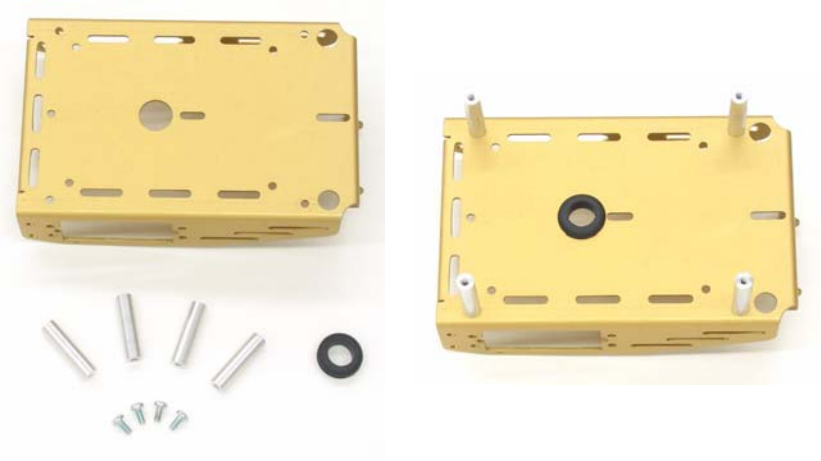


Figure 3-2
Chassis and Topside Hardware

Parts (left); assembled (right).

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. See Appendix E: Boe-Bot Parts Lists for more information.

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.

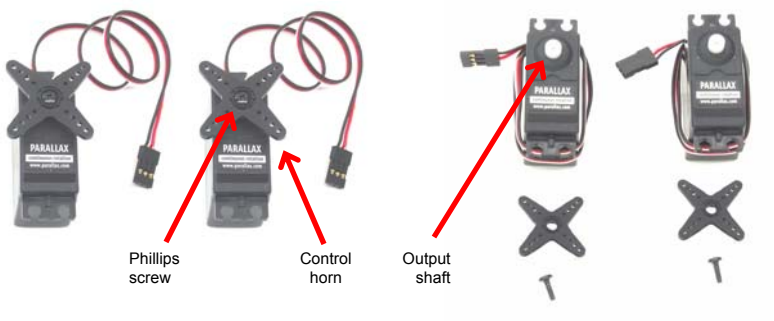


Figure 3-3
Servo Control Horn Removal

Parts (left); after following instructions (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
- Activity #4: Centering the Servos

Mounting the Servos on the Chassis

Parts List:

See Figure 3-4.

(1) 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. Note that for best performance, you must place the face of each servo through the rectangular window from inside the chassis rather than dropping them in from the outside.

√ Use pieces of masking tape to label the servos left (L) and right (R).

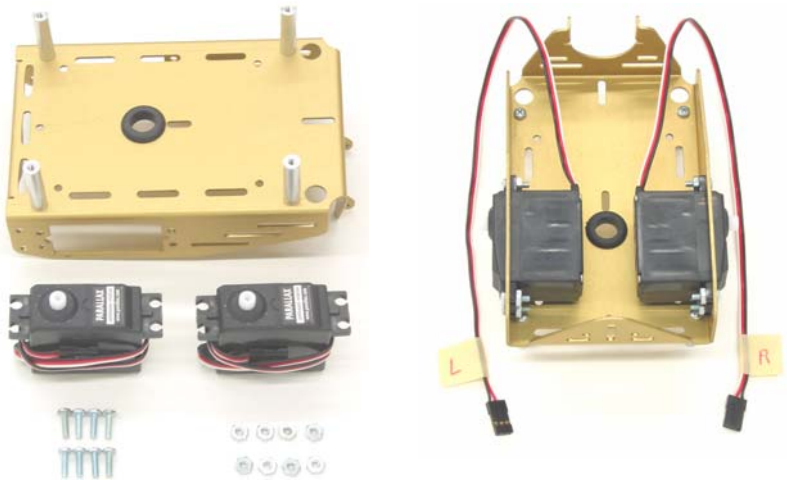


Figure 3-4
Mounting the Servos on the Chassis

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 Rev C:

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

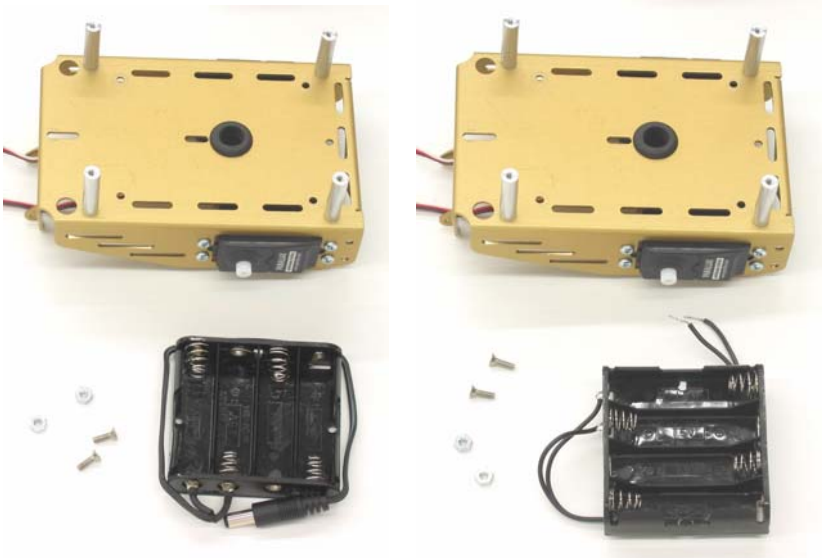


Figure 3-5
Battery Pack Mounting Hardware

For use with the Board of Education

For use with the HomeWork Board

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, 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.

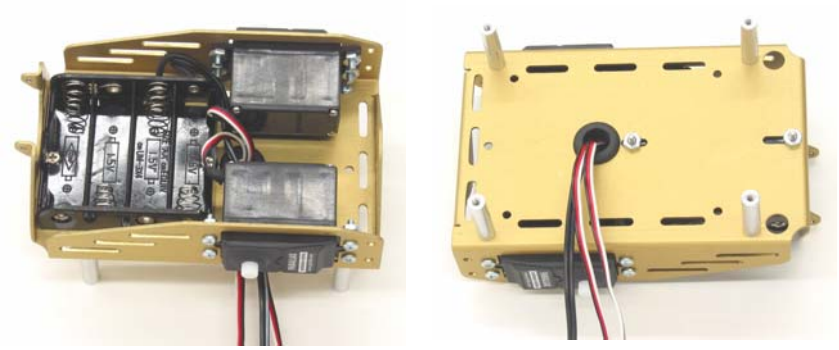


Figure 3-6
Battery Pack Installed

Bottom view (left); top view (right).

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.

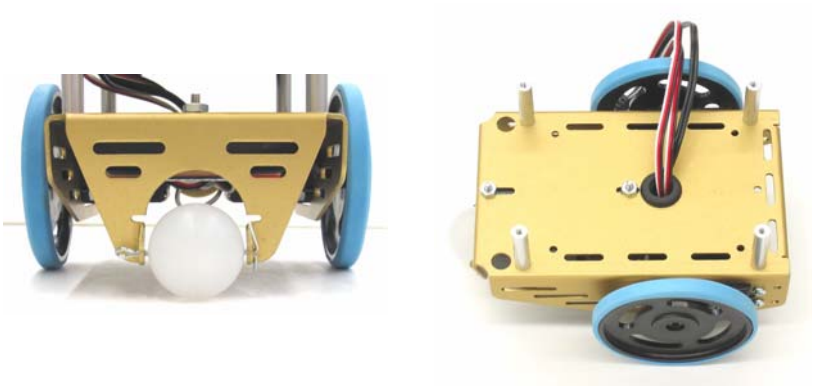


Figure 3-8
Mounting the Wheels

Tail wheels (left); drive wheels (right).

Attaching Board to Chassis

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

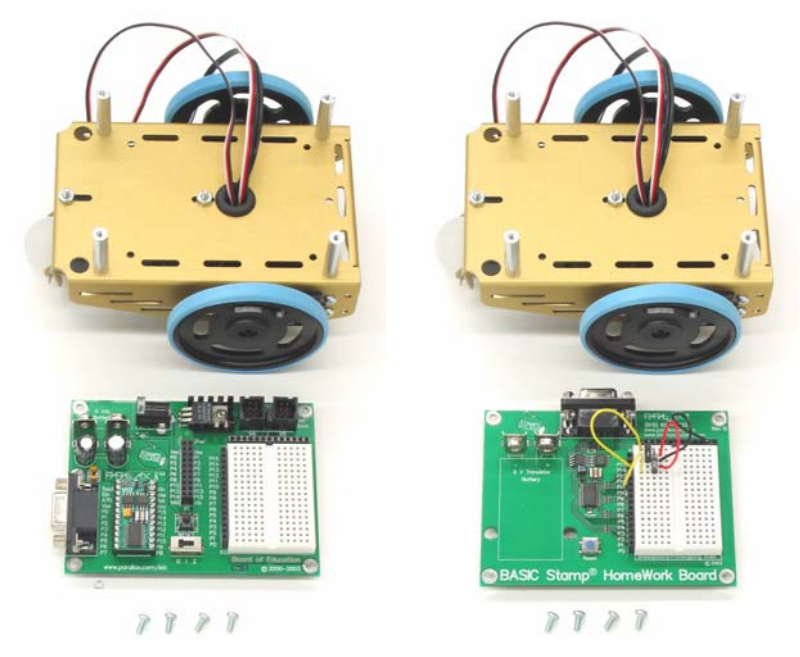
See left side of Figure 3-9.

(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 a 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



With the Board of Education Rev C With the HomeWork Board

Figure 3-9
Boe-Bot Chassis and Boards

Figure 3-10 shows the servo ports reconnected for both the Board of Education Rev C (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.

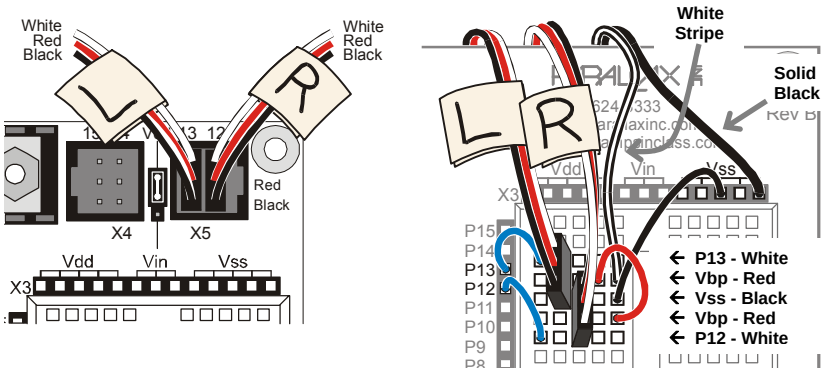


Figure 3-10 Servo Ports Reconnected

Board of Education Rev C (left) HomeWork Board (right).

On Board of Education Rev C On HomeWork Board

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.

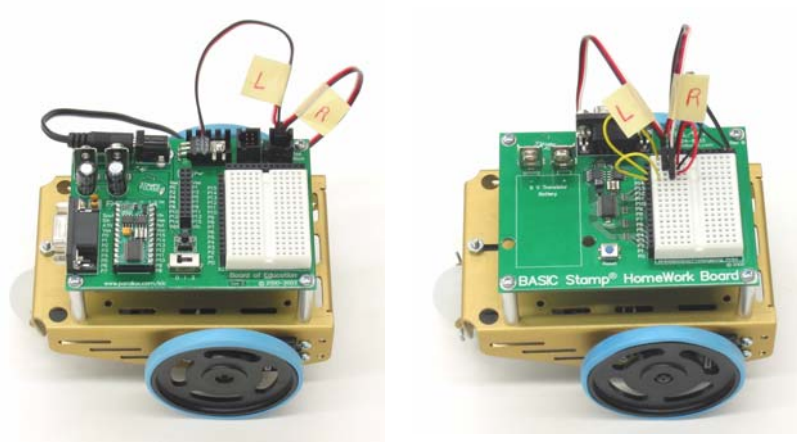


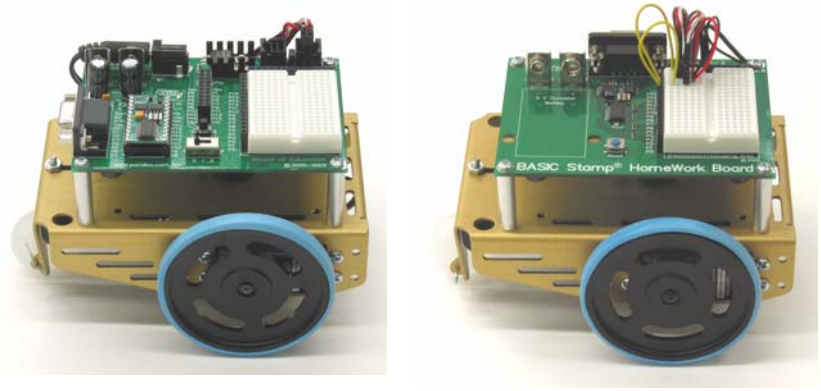
Figure 3-11 Boards Attached to Boe-Bot Chassis

With Board of Education Rev C With HomeWork Board

Figure 3-12 shows assembled Boe-Bots, the left built with a Board of Education 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 Rev C With HomeWork Board

Figure 3-12 Assembled Boe-Bots

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.

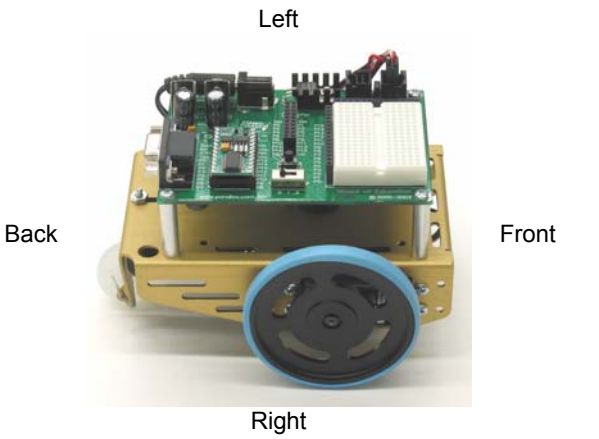


Figure 3-13
Your Boe-Bot robot's Front, Back, Left, and Right

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.

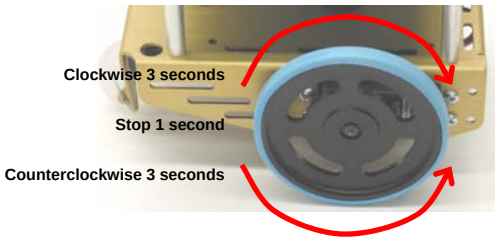


Figure 3-14
Testing the Right Wheel

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 Rev C, 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 Trouble Shooting 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
```

Servo Trouble Shooting: 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 Rev C, 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 batteries.
- √ Double-check your servo connections using Figure 3-10 on page 100 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 65.
- √ 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. You can often adjust the program to make the servo stay still. You can do this by modifying the **`PULSOUT 12, 750`** command.

- √ 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.
- √ If you can find a value between 740 and 760 that fully stops your servo, then make sure to use it anywhere you see the command **`PULSOUT 12, 750`**.

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 Chapter 2 Activity #4: Centering the Servos.

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`.

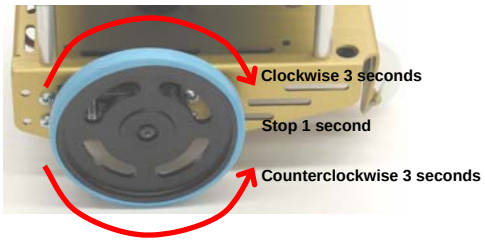


Figure 3-15
Testing the Left Wheel

- √ 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 Trouble Shooting section on page 104.
- √ 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.

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 the 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 1.5 V alkaline 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.

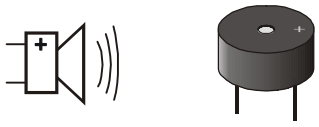


Figure 3-16
Piezospeaker

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.

Note: 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 Rev C, 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.

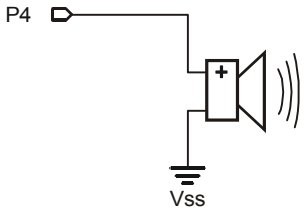


Figure 3-17
Program Start/Reset Indicator Circuit

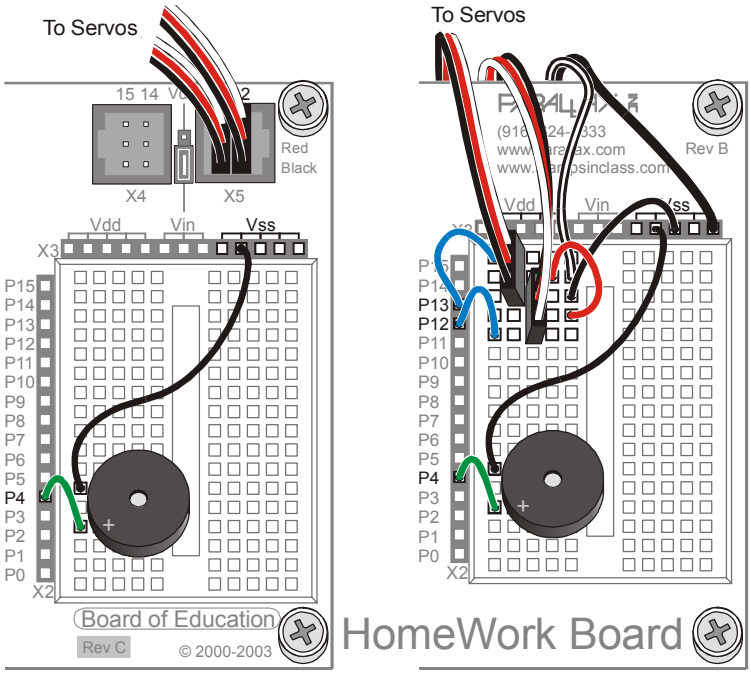


Figure 3-18
Wiring Diagrams for the Program Start/Reset Indicator Circuit

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

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.

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.

- √ 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.
```

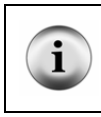
How StartResetIndicator.bs2 Works

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 lines of code 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.

3

- √ Open HelloOnceEverySecond.bs2.
- √ Copy the **FREQOUT** command from StartResetIndicator.bs2 into HelloOnceEverySecond.bs2 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.

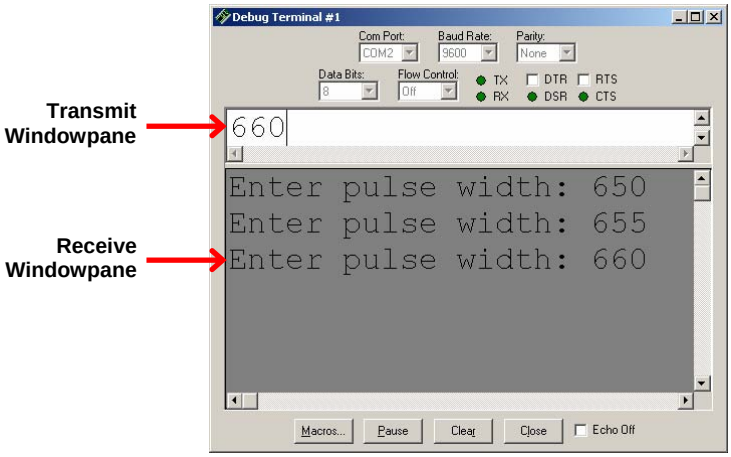


Figure 3-19
Debug Terminal Windowpanes

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 that last longer are wider on timing diagrams, and pulses that 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.

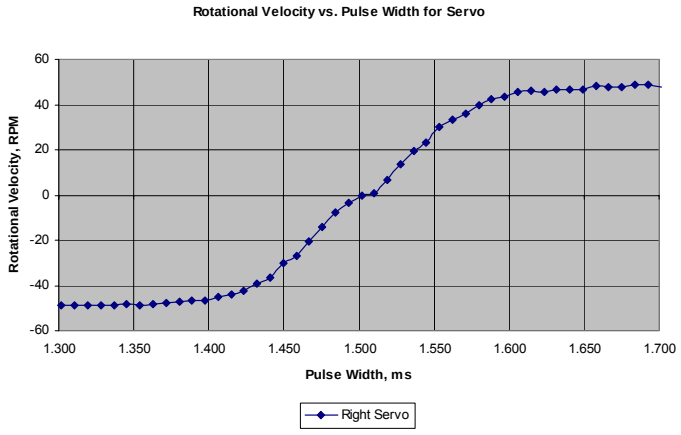


Figure 3-20
Transfer Curve Example for Parallax Servo

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.

Table 3-1: Pulse Width and RPM for Parallax 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	

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.

$$\begin{aligned} Duration &= 650 \times 2\ \mu s \\ &= 650 \times 0.000002\ s \\ &= 0.0013\ s \\ &= 1.3\ ms \end{aligned}$$

$$\begin{aligned} Duration &= 655 \times 2\ \mu s \\ &= 655 \times 0.000002\ s \\ &= 0.00131\ s \\ &= 1.31\ ms \end{aligned}$$

$$\begin{aligned} Duration &= 660 \times 2\ \mu s \\ &= 660 \times 0.000002\ s \\ &= 0.00132\ s \\ &= 1.32\ ms \end{aligned}$$

- √ Mark your right wheel so that you have a reference point to count the revolutions.
- √ Run TestServoSpeed.bs2.
- √ 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 into Table 3-1 next to the 1.3 ms entry.
- √ Enter the value 655.
- √ Count how many turns the wheel made.
- √ Multiply this value by 10 and enter the result into Table 3-1 next to the 1.31 ms 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.

You can repeat these measurements for the left wheel. You will have to 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 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.

```
' 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
```

Note: This project is best tested with the Boe-Bot's wheels propped up.

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 straight line, 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.

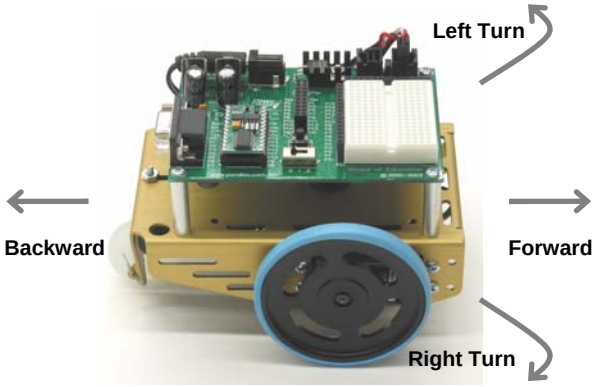


Figure 4-1
Your Boe-Bot and Driving Directions

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.

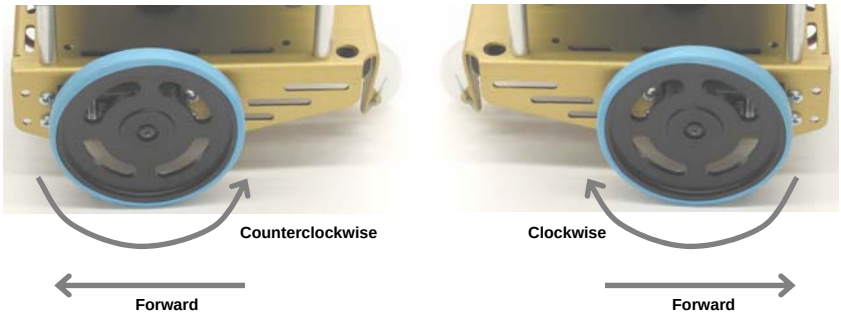


Figure 4-2
Wheel Rotation for Forward Motion

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 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 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.

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

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.

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

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:

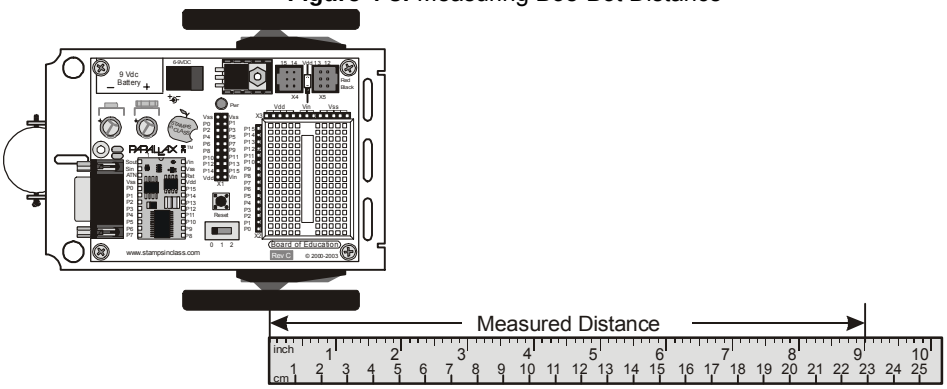
$$servo\ run\ time = \frac{Boe\text{-}Bot\ distance}{Boe\text{-}Bot\ speed}$$

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 in

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

$$time = \frac{20\ in}{9\ in/s}$$
$$= 20\ in \times \frac{1\ s}{9\ in}$$
$$= 2.22\ s$$

Example – Time for 51 cm

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

$$time = \frac{51\ cm}{23\ cm/s}$$
$$= 51\ cm \times \frac{1\ s}{23\ cm}$$
$$= 2.22\ s$$

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.

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

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 in

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 cm

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

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

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. Encoders hardware, documentation and example programs are available in the Robotics → Accessories page at www.parallax.com.

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
```

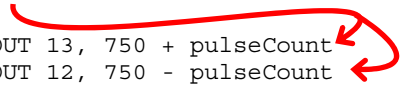


Figure 4-4 Ramping Example

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".

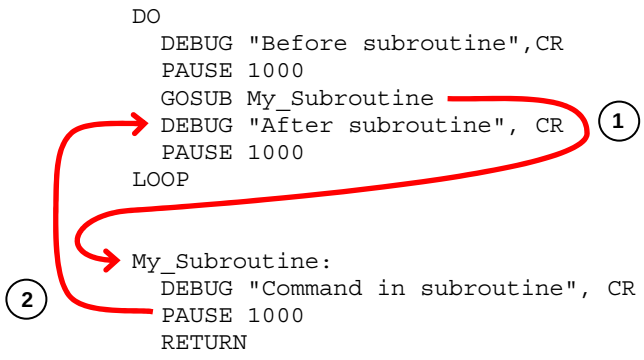


Figure 4-5 Subroutine Basics

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 132.

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

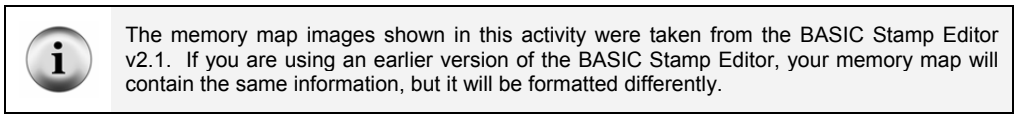
When you download PBASIC program to your BASIC Stamp, the BASIC Stamp Editor converts your program to numeric values called tokens. These tokens are what the BASIC Stamp uses as instructions for executing the program. They are stored in one of the two smaller black chips on top of your BASIC Stamp, the one labeled "24LC16B." 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 an earlier version of the BASIC Stamp Editor, your memory map will contain the same information, but it will be formatted differently.

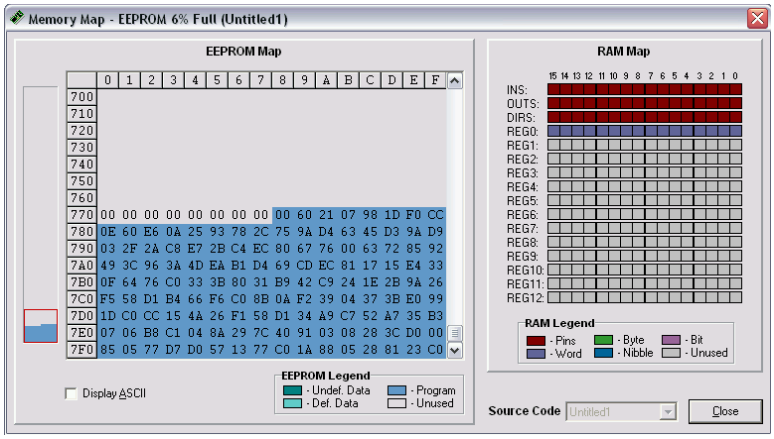


Figure 4-6
BASIC Stamp Memory Map

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 (American Standard Code for Information Interchange) codes that correspond to the characters you entered in your **DATA** statement.

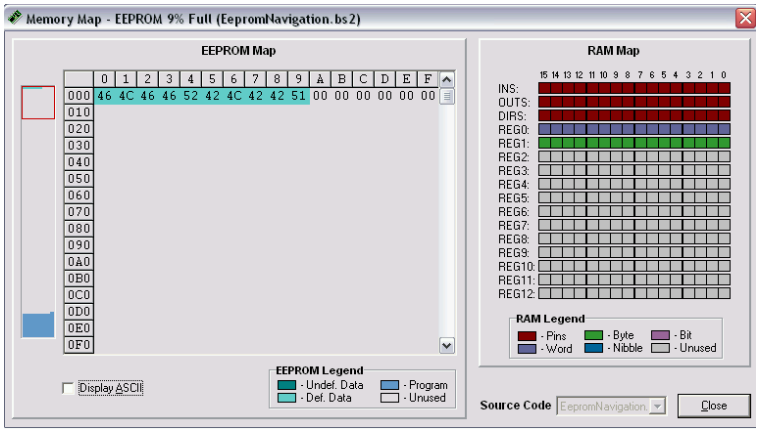


Figure 4-7 Memory Map with Stored Instructions Visible in EEPROM Map

√ 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.

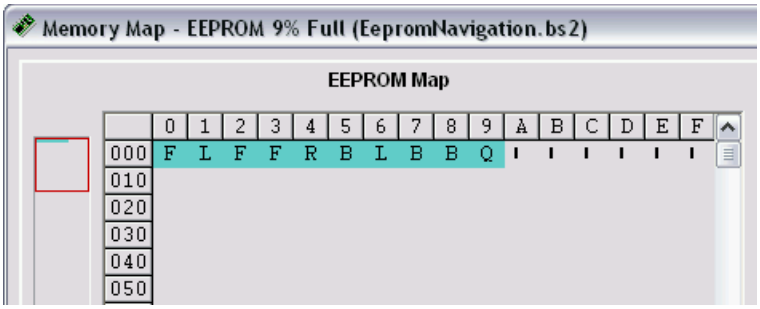


Figure 4-8
Close-up of the Detailed EEPROM Map after Display ASCII Box is Checked

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 on page 153. 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 153 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: **`PULSOUT` *`Duration`*** Combinations on page 81. 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 obstacle 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.

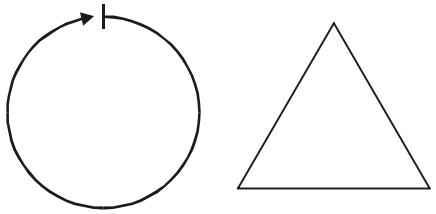


Figure 4-9
Simple Obstacle Courses

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:

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

It should take 133 pulses to travel 36 inches.

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, 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.

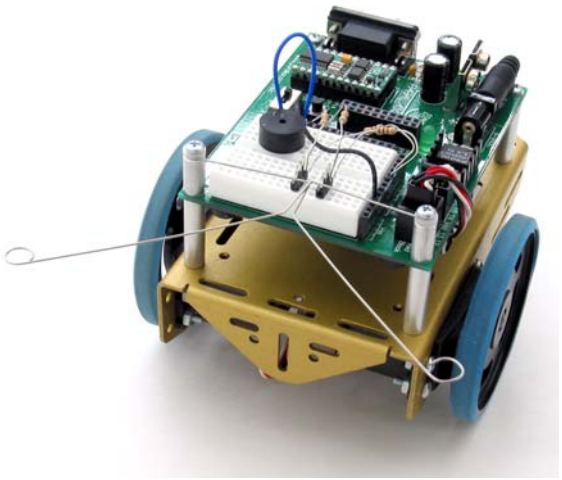


Figure 5-1
Boe-Bot with Whiskers

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

5

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 the washer and the other below the washer, positioning them so they cross over each other without touching.
- √ Tighten the screws into the standoffs.

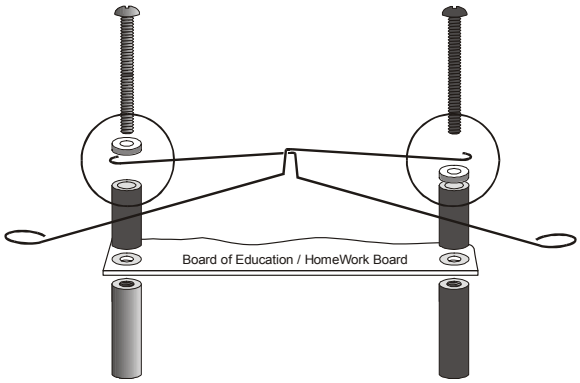


Figure 5-3 Mounting the Whiskers

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 169 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 170 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.

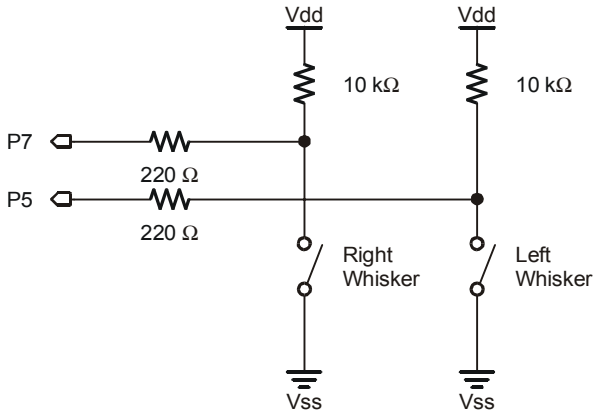
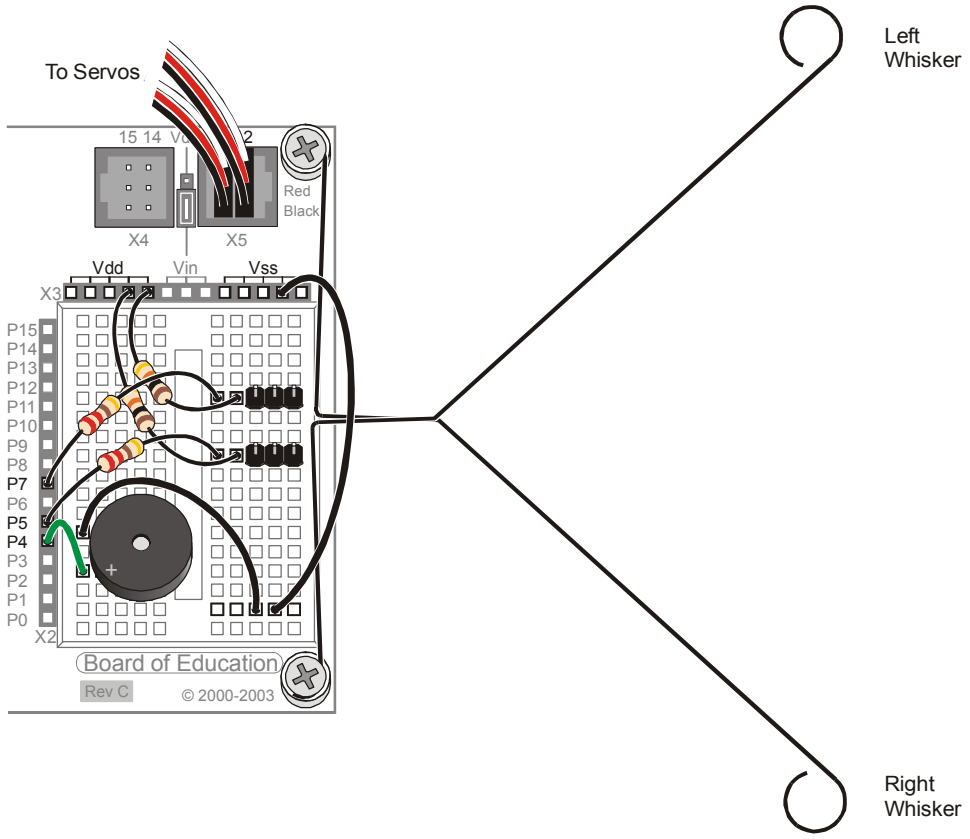


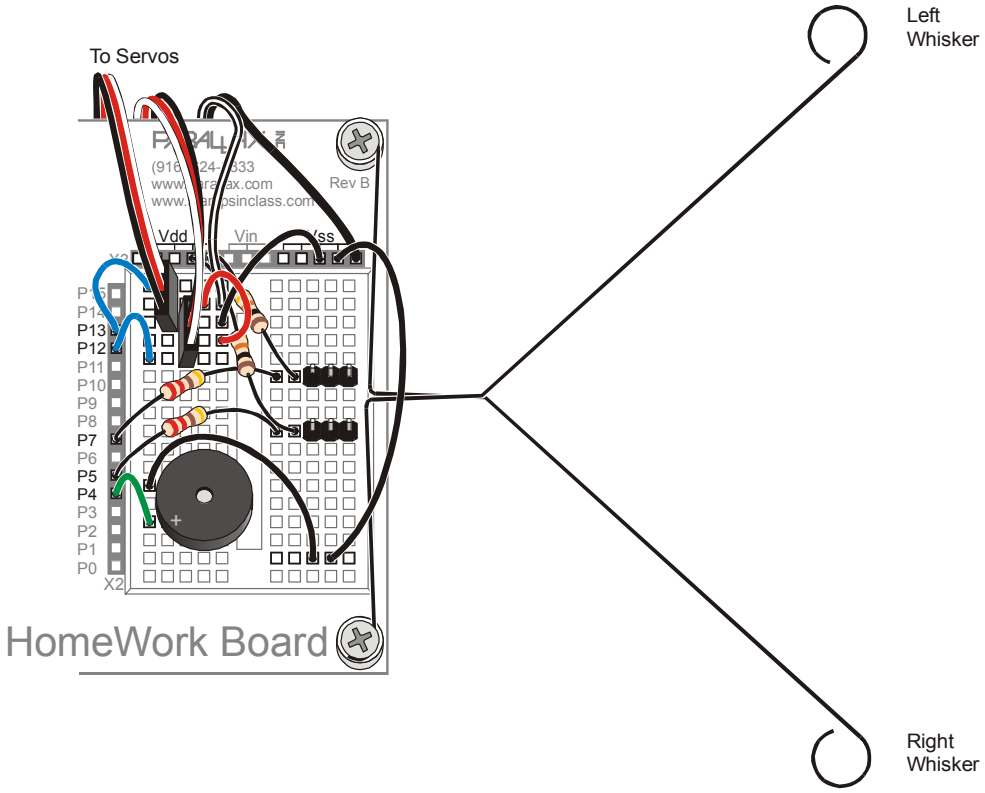
Figure 5-4
Whiskers
Schematic

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.

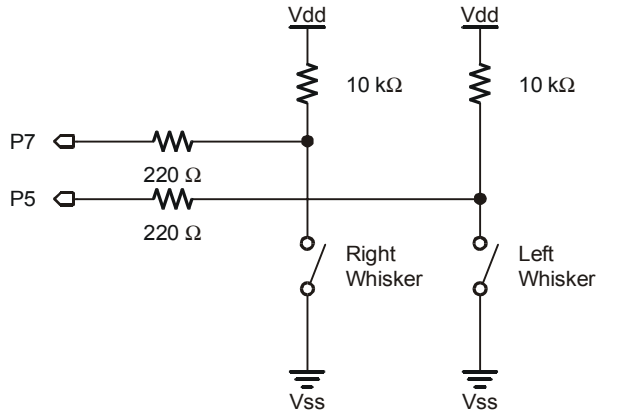


Figure 5-7 Whiskers Schematic – A Second Look

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).

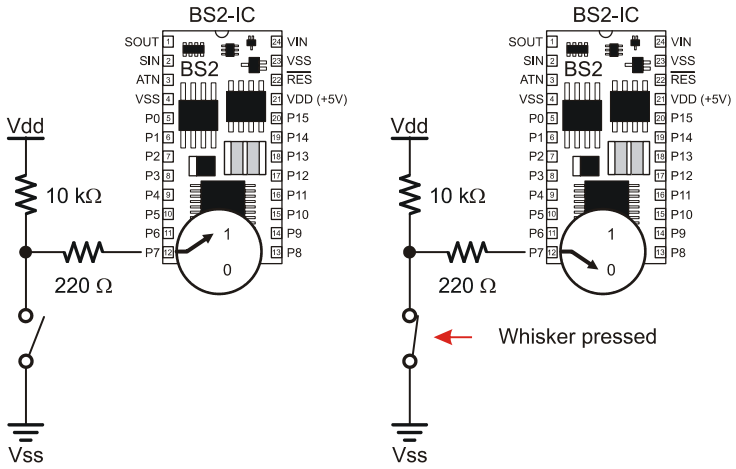


Figure 5-8
Detecting Electrical Contacts

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 serial 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 169 (or Figure 5-6 on page 170) so you know which whisker is the "left whisker" and which whisker 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 `CRSRXY`?

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 175).
- √ 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 176).

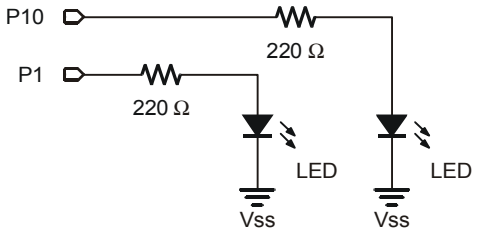
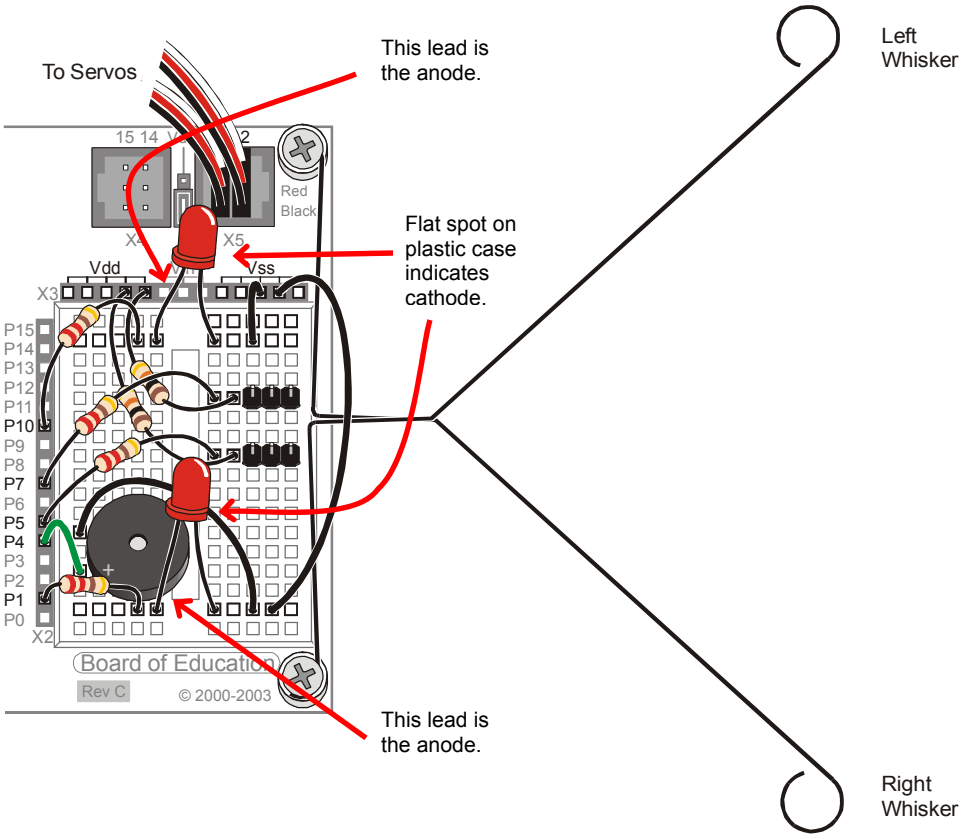


Figure 5-9
LED Whisker Testing Schematic

Add this LED circuit.

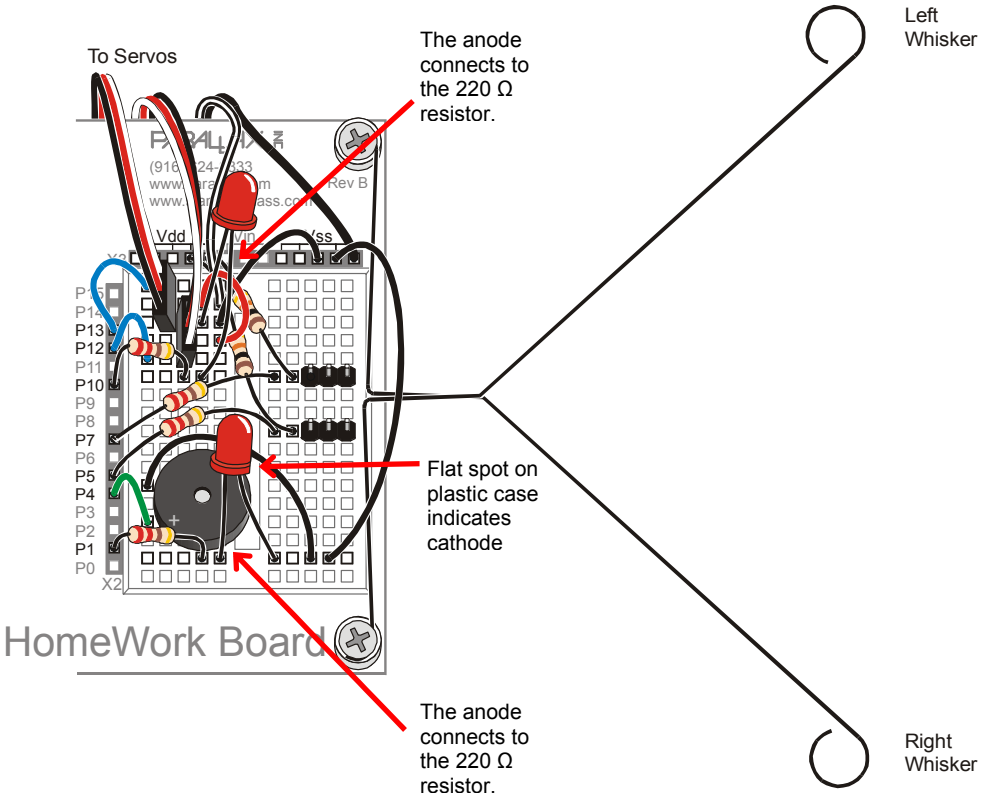
Remember that an LED is a one way current valve. If it is plugged in backwards, it will not let current pass through, and so will not emit light. For the LED to emit light when the BASIC Stamp sends a high signal, the LED's anode must be connected to the 220 Ω resistor, and its cathode must be connected to Vss. See Figure 5-10 or Figure 5-11.

Figure 5-10: Whisker Plus LED Wiring Diagram for the Board of Education



5

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 Activity #1, 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 the 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 the 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 an 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.
```
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 1y 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.

In the solution below, an audible beeping indicator was added. This acts as feedback to verify that the whisker was pressed. This is entirely optional.

```
' 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 Photoresistors

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 sensor in your Boe-Bot kit is designed to detect visible light, and it can be used to make your Boe-Bot detect variations in light level. With this ability, your Boe-Bot can be programmed to recognize areas with light or dark perimeters, report the overall brightness and darkness level it sees, and seek out light sources such as flashlight beams and doorways letting light into dark rooms.

INTRODUCING THE PHOTORESISTOR

The resistors you worked with in previous chapters had fixed values, such as 220 Ω and 10 kΩ. The photoresistor, on the other hand, is a light dependent resistor (LDR). This means that its resistance value depends on the brightness, or illuminance, of the light that shines on its light detecting surface. Figure 6-1 shows the schematic symbol and part drawing for the photoresistor you will use to make the Boe-Bot able to detect variations in light levels.

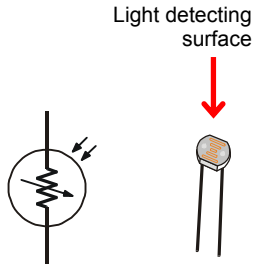


Figure 6-1
Photoresistor Schematic and Part Drawing

A photoresistor is a light-dependent resistor (LDR) that covers the spectral sensitivity similar to that of the human eye. In other words, the kind of light that your eye detects is the same kind of light that affects the photoresistor's resistance. The active elements of these photoresistors are made of Cadmium Sulfide (CdS). Light enters into the semiconductor layer applied to a ceramic substrate and produces free charge carriers. A defined electrical resistance is produced that is inversely proportional to the illumination intensity. In other words, darkness causes more resistance, and brightness causes less resistance.

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 shining on the wall is incident light. The unit of measurement of illuminance is commonly the "foot-candle" in the English system or the "lux" in the metric system. While using the photoresistors we won't be concerned about lux levels, just whether illuminance is higher or lower in certain directions. The Boe-Bot can be programmed to use the relative light intensity information to make navigation decisions.

ACTIVITY #1: BUILDING AND TESTING PHOTORESISTOR CIRCUITS

In this activity, you will build and test light level sensor circuits with photoresistors. Your light level sensor circuits will be able to detect the difference between shade and no shade. The PBASIC commands for determining whether a shadow is cast over the photoresistor will be very similar to those used to determine whether or not a whisker has contacted an object.

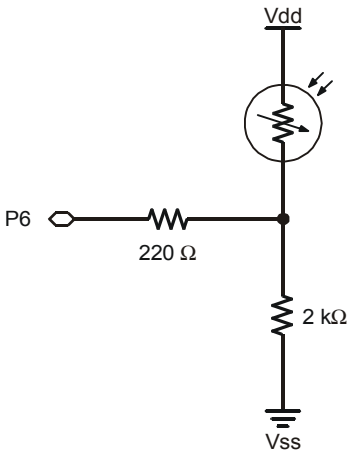
Parts List:

(2) Photoresistors - CdS
(2) Resistors – 2 kΩ (red-black-red)
(2) Resistors – 220 Ω (red-red-brown)
(4) Jumper wires
(2) Resistors – 470 Ω (yellow-violet-brown)
(2) Resistors – 1 kΩ (brown-black-red)
(2) Resistors – 4.7 kΩ (yellow-violet-red)
(2) Resistors – 10 kΩ (brown-black-orange)

Building the Photosensitive Eyes

Figure 6-2 shows the schematic and Figure 6-3 shows the wiring diagram for the photoresistor circuits you will use in this and the next two activities.

√ Disconnect power from your board and servos.
√ Build the circuit shown in Figure 6-2, using Figure 6-3 as a reference.



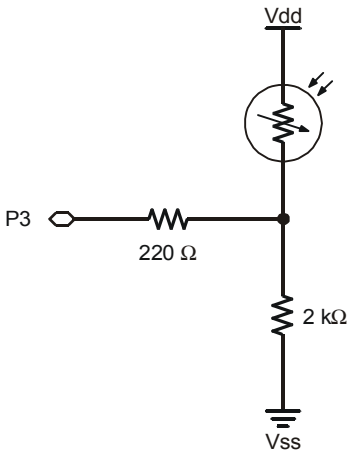
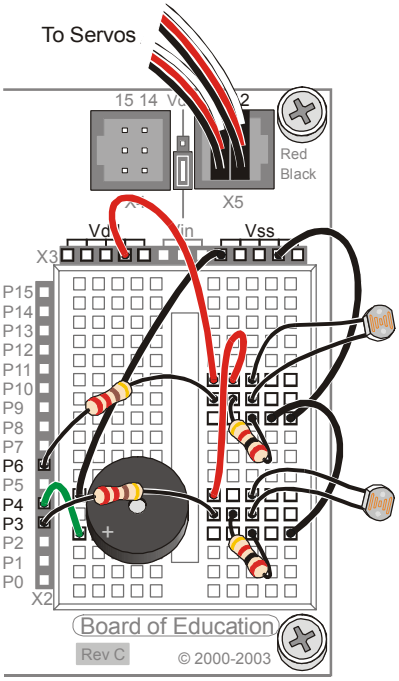


Figure 6-2
Schematic – First Light Detection Circuit



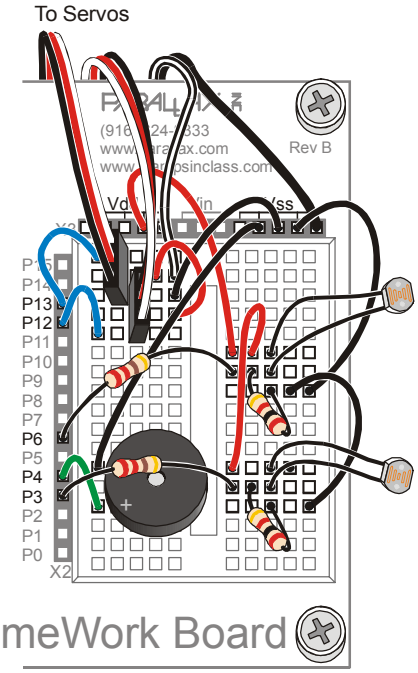


Figure 6-3
Wiring Diagrams for the First Light Detection Circuit

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

How the Photoresistor Circuit Works

A BASIC Stamp I/O pin can function as an output or an input. As an output, the I/O pin can send a high (5 V) or low (0 V) signal. Up to this point, high and low signals have been used to turn LED circuits on and off, control servos, and send tones to a speaker.

A BASIC Stamp I/O pin can also function as an input. As an input, the I/O pin does not apply any voltage to the circuit it is connected to. Instead, it just quietly listens without any actual effect on the circuit. In the previous chapter, these input registers stored values that indicated whether or not the whiskers were pressed. For example, the **IN7** input register stored a 1 when it sensed 5 V (whisker not pressed), or a 0 when it sensed 0 V (whisker pressed).

An I/O pin set to input doesn't actually need to have 5 V applied to it to make its input register store a 1. Anything above 1.4 V will make the input register for an I/O pin store a 1. Likewise, an I/O pin doesn't need 0 V to make its input register store a 0. Any voltage below 1.4 V will make an input register for an I/O pin store a 0.

BASIC Stamp I/O pins are input by default. When a BASIC Stamp program starts, all I/O pins start as inputs. When you use commands like **HIGH**, **LOW**, **PULSOUT** or **FREQOUT**, the I/O pin is changed from input to output so that the BASIC Stamp can send the high or low signals.

When a BASIC Stamp I/O pin is an input, the circuit behaves as though neither the I/O pin nor 220 Ω resistor is present. Figure 6-4 shows the equivalent circuit. The resistance of the photoresistor is shown as the letter R. It could be a few Ω if the light is very bright, or it could be in the neighborhood of 50 kΩ in complete darkness. In a well lit room with fluorescent ceiling fixtures, the resistance could be as small as a 1 kΩ (full light exposure) or as large as 25 kΩ (shade cast around most of the object).

As the photoresistor's resistance changes with light exposure, so does the voltage at Vo; as R gets larger, Vo gets smaller, and as R gets smaller, Vo gets larger. Vo is what the BASIC Stamp I/O pin is detecting when it is functioning as an input. If this circuit is connected to **IN6**, when the voltage at Vo is above 1.4 V, **IN6** will store a 1. If Vo falls below 1.4 V, **IN6** will store a 0.

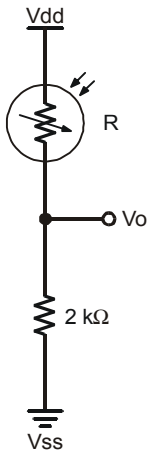


Figure 6-4
Schematic – Voltage Divider Circuit

When resistors are connected end-to-end as shown in Figure 6-4 they are **connected in series**, and they can be referred to as series resistors.

When two resistors are connected in series to set a voltage at Vo, the circuit is called a **voltage divider**. In this circuit, the value of Vo can be anywhere between Vdd and Vss. The exact value of Vo is determined by the ratio of R to 2 kΩ. When R is larger than 2 kΩ, Vo will be closer to Vss. When R is smaller than 2 kΩ, Vo will be closer to Vdd. When R is equal to 2 kΩ, Vo will be 2.5 V. If you measure one of the two values (R or Vo), you can calculate the other value using one of these two equations.

$$Vo = 5V \times \frac{2000\Omega}{2000\Omega + R} \qquad R = \left(5V \times \frac{2000\Omega}{Vo} \right) - 2000\Omega$$

1.4 V is called the BASIC Stamp I/O pin's **threshold voltage**, also known as the I/O pin's **logic threshold**. When voltage sensed by an I/O pin is above that threshold, the I/O pin's input register will store a 1. If it is below that value, the I/O pin's input register will store a 0.

Detecting Shadows

Casting a shadow makes the photoresistor's resistance value (R) larger, which in turn makes Vo smaller. The 2 kΩ resistors were chosen to make the value of Vo reside slightly above the BASIC Stamp I/O pin's 1.4 V threshold in a well lit room. When you cast a shadow over it with your hand, it should send Vo below the 1.4 V threshold.

In a well lit room, both `IN6` and `IN3` will store the value 1. If you cast a shadow over the photoresistor divider connected to P6, it will then store a 0. Likewise, if you cast a shadow over the photoresistor divider connected to P3, it will cause `IN3` to store a 0.

Example Program: TestPhotoresistorsDividers.bs2

This example program is TestWhiskers.bs2 adapted to the photoresistor dividers. Instead of monitoring P5 and P7 as we did with the whiskers, we are now monitoring P3 and P6, which are connected to the photoresistor divider circuits. This program should display a value of 1 on both sides in a well-lit room. When you cast a shadow over one or both of the photoresistors, their corresponding values should change to 0.

- √ Reconnect power to your board.
- √ Enter, save, and run TestPhotoresistorDividers.bs2.
- √ Verify that with no shade, both **IN6** and **IN3** store the value 1.
- √ Verify that you can use your hand to cast a shadow over each of the photoresistors and cause its input register to change from 1 to 0.
- √ If you are having difficulty, either with getting a shadow to change the input register to 0 or if the input registers store 0 regardless of whether or not you cast a shadow over them, see the Photoresistor Divider Troubleshooting box after the program listing. Work on it until your hand casting a shadow over the photoresistor reliably changes the state from 1 to 0.

```
' Robotics with the Boe-Bot - TestPhotoresistorDividers.bs2
' Display what the I/O pins connected to the photoresistor
' voltage dividers sense.

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

DEBUG "PHOTORESISTOR STATES", CR,
      "Left       Right", CR,
      "-------    --------"

DO
  DEBUG CRSRXY, 0, 3,
        "P6 = ", BIN1 IN6,
        "     P3 = ", BIN1 IN3
  PAUSE 100
LOOP
```

Photoresistor Divider Troubleshooting

General things to verify:

- √ Check your wiring and program for errors.
- √ Make sure that each component is firmly plugged into its socket.
- √ Check the color codes on your resistors. The resistors that connect between Vss and the photoresistors should be 2 kΩ (red-black-red). The resistors connecting P6 and P3 to the photoresistors should be 220 Ω (red-red-brown).

If either the IN3 or IN6 registers showed 0 regardless of whether or not you cast a shadow over them:

- √ If the room is dimly lit, consider bringing in some extra lamps. Alternately, you can replace the 2 kΩ resistors with 4.7 kΩ resistors (Yellow Violet Red). This will give your resistor divider better performance under lower lighting conditions. For really low lighting conditions, you can even use the 10 kΩ resistors (brown-black-orange).

If either the IN3 or IN6 registers showed 1 regardless of whether or not you cast a shadow over them:

- √ If the room is very brightly lit, and you find yourself having to cup your hand over the photoresistor's light collecting surface to make the 1 go to 0, you may need to substitute a lower value resistor in place of the 2 kΩ. Try 1 kΩ resistor (brown-black-red), or even a 470 Ω resistor (yellow-violet-brown) if you are outdoors.

Your Turn – Experimenting with Different Voltage Dividers

Depending on the lighting conditions in your robotics area, larger or smaller series resistors in place of the 2 kΩ resistors may improve the performance of your shadow detectors.

- √ Remember to disconnect power to your board during each circuit modification.
- √ Try replacing the 2 kΩ (red-black-red) resistors with each of the other resistor values you have gathered: 470 Ω, 1 kΩ, 4.7 kΩ, and 10 kΩ.
- √ Test each voltage divider combination with TestPhotoresistorDividers.bs2 and determine which resistors work best under your lighting conditions. The best combination is one that's not overly sensitive, but doesn't require you to cup your hand over the photoresistor either.
- √ Use the resistor combination that you think works best in the next two activities.

ACTIVITY #2: ROAM AND AVOID SHADOWS LIKE OBJECTS

Since the photoresistor dividers behave similarly to whiskers, it's worth examining what's involved in adapting RoamingWithWhiskers.bs2 so that it functions with the photoresistor dividers.

Adapting RoamingWithWhiskers.bs2 for the Photoresistor Dividers

All you really have to do is adjust the **IF…THEN** statements so that they monitor **IN6** and **IN3**, instead of **IN7** and **IN5**. Figure 6-5 demonstrates how to make these changes.

Figure 6-5: Modify RoamingWithWhiskers.bs2 for Use with Photoresistor Dividers

```
' From RoamingWithWhiskers.bs2

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
```

```
' Modified for
' RoamingWithPhotoresistor
' Dividers.bs2

IF (IN6  = 0) AND (IN3 = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Left
  GOSUB Turn_Left
ELSEIF (IN6  = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Right
ELSEIF (IN3  = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Left
ELSE
  GOSUB Forward_Pulse
ENDIF
```

Example Program – RoamingWithPhotoresistorDividers.bs2

- √ Open the program RoamingWithWhiskers.bs2 from page 179, and save it as RoamingWithPhotoresistorDividers.bs2.
- √ Make the modifications shown in Figure 6-5.
- √ Reconnect power to your board and servos.
- √ Run and test the program.

Casting shadows over both photoresistors at the same time can be difficult. When the Boe-Bot is going forward, it is checking the photoresistors around 40 times/second. You will have to move quickly to cast a shadow over both photoresistors between pulses. It helps to move your hand rapidly from no shade to full shade to trigger both photoresistors at once. Alternately, just leave your hand casting shade over both photoresistors while it executes a maneuver. When it returns from the maneuver and checks the photoresistors again, it should recognize that both photoresistors are in shade.□

- √ Verify that the Boe-Bot avoids shadows by using your hand to cast a shadow over the photoresistors. Try no shadow, a shadow over the right photoresistor divider (circuit connected to P3), a shadow over the left photoresistor divider (circuit connected to P7), and a shadow over both photoresistor dividers.
- √ Update the comments such as the title and descriptions of reactions to whisker presses to reflect the photoresistor circuit behavior. It should resemble the program below when you are done.


```
' -----[ Title ]-----------------------------------------------------------
' Robotics with the Boe-Bot - RoamingWithPhotoresistorDividers.bs2
' Boe-Bot detects shadows photoresistors voltage divider circuit and turns
' away from 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                      ' Start/restart signal.

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

DO
IF (IN6  = 0) AND (IN3 = 0) THEN           ' Both photoresistors detects
  GOSUB Back_Up                            ' shadow, back up & U-turn
  GOSUB Turn_Left                          ' (left twice).
  GOSUB Turn_Left
ELSEIF (IN6  = 0) THEN                     ' Left photoresistor detects
  GOSUB Back_Up                            ' shadow, back up & turn right.
  GOSUB Turn_Right
ELSEIF (IN3  = 0) THEN                     ' Right photoresistor detects
  GOSUB Back_Up                            ' shadow, back up & turn left.
  GOSUB Turn_Left
```

```
ELSE                                          ' Neither photoresistor detects
  GOSUB Forward_Pulse                         ' shadow, apply a forward pulse.
ENDIF

LOOP

' -----[ Subroutines ]-------------------------------------------------------

Forward_Pulse:                                ' Send a single forward pulse.
  PULSOUT 12,650
  PULSOUT 13,850
  PAUSE 20
  RETURN

Turn_Left:                                    ' Left turn, about 90-degrees.
  FOR pulseCount = 0 TO 20
    PULSOUT 12, 650
    PULSOUT 13, 650
    PAUSE 20
  NEXT
  RETURN

Turn_Right:
  FOR pulseCount = 0 TO 20                    ' Right turn, about 90-degrees.
    PULSOUT 12, 850
    PULSOUT 13, 850
    PAUSE 20
  NEXT
  RETURN

Back_Up:                                      ' Back up.
  FOR pulseCount = 0 TO 40
    PULSOUT 12, 850
    PULSOUT 13, 650
    PAUSE 20
  NEXT
  RETURN
```

Your Turn – Improving performance

You can improve your Boe-Bot's performance by commenting some of the subroutine calls that were designed to help the Boe-Bot back away from obstacles and then turn to avoid them. Figure 6-6 shows an example where the two **`Turn_Left`** subroutine calls are commented from the **`IF…THEN`** statement when the condition is that both photoresistors detect a shadow. Then, when only individual photoresistors detect shadows, the **`Back_Up`** subroutine calls are commented so that the Boe-Bot only turns in response to a shadow.

Figure 6-6: Modify RoamingWithPhotoresistorDividers.bs2

```
' Excerpt from                          ' Modified excerpt from
' RoamingWithPhotoresistor              ' RoamingWithPhotoresistor
' Dividers.bs2                         ' Dividers.bs2

IF (IN6  = 0) AND (IN3 = 0) THEN       IF (IN6  = 0) AND (IN3 = 0) THEN
  GOSUB Back_Up                          GOSUB Back_Up
  GOSUB Turn_Left                      '   GOSUB Turn_Left
  GOSUB Turn_Left                      '   GOSUB Turn_Left
ELSEIF (IN6  = 0) THEN                 ELSEIF (IN6  = 0) THEN
  GOSUB Back_Up                        '   GOSUB Back_Up
  GOSUB Turn_Right                       GOSUB Turn_Right
ELSEIF (IN3  = 0) THEN                 ELSEIF (IN3  = 0) THEN
  GOSUB Back_Up                        '   GOSUB Back_Up
  GOSUB Turn_Left                        GOSUB Turn_Left
ELSE                                   ELSE
  GOSUB Forward_Pulse                    GOSUB Forward_Pulse
ENDIF                                  ENDIF
```

- √ Modify RoamingWithPhotoresistorDividers.bs2 as shown in the right side of Figure 6-6.
- √ Run the program, and compare the performance.

ACTIVITY #3: A MORE RESPONSIVE SHADOW CONTROLLED BOE-BOT

By eliminating the **FOR…NEXT** loops in the navigation subroutines, you can make the Boe-Bot significantly more responsive. This wasn't really possible with the whiskers, because the Boe-Bot had to back up before turning since it had already made physical contact with the obstacle. When you are using shadows to guide the Boe-Bot, it can check between each pulse to see if the shadow is still detected regardless of whether it's moving forward or executing a maneuver.

A Simple Shadow Controlled Boe-Bot

One interesting form of remote control is to have the Boe-Bot sit still in normal light, then follow a shadow you cast over the photoresistors. It's kind of a user-friendly way of guiding the Boe-Bot's motion.

Example Program – ShadowGuidedBoeBot.bs2

When you run this next program, the Boe-Bot should stay still when no shadow is cast over its photoresistor dividers. When you cast a shadow over both photoresistors, the

Boe-Bot should move forward. If you cast a shadow over one of the photoresistors, the Boe-Bot should turn in the direction of the photoresistor that senses the shadow.

- √ Enter, save, and run ShadowGuidedBoeBot.bs2.
- √ Use your hand to cast shadows over the photoresistor dividers.
- √ Study this program carefully and make sure you understand how it works. It is very short, yet very powerful.

```
' Robotics with the Boe-Bot - ShadowGuidedBoeBot.bs2
' Boe-Bot detects shadows cast by your hand and tries to follow them.

' {$STAMP BS2}                                ' Stamp directive.
' {$PBASIC 2.5}                               ' PBASIC directive.

DEBUG "Program Running!"
FREQOUT 4, 2000, 3000                         ' Start/restart signal.

DO

  IF (IN6  = 0) AND (IN3 = 0) THEN            ' Both detect shadows, forward.
    PULSOUT 13, 850
    PULSOUT 12, 650
  ELSEIF (IN6  = 0) THEN                      ' Left detects shadow,
    PULSOUT 13, 750                           ' pivot left.
    PULSOUT 12, 650
  ELSEIF (IN3  = 0) THEN                      ' Right detects shadow,
    PULSOUT 13, 850                           ' pivot right.
    PULSOUT 12, 750
  ELSE
    PULSOUT 13, 750                           ' No shadow, sit still
    PULSOUT 12, 750
  ENDIF

  PAUSE 20                                    ' Pause between pulses.

LOOP
```

How ShadowGuidedBoeBot.bs2 Works

The **IF…THEN** statement in the **DO…LOOP** looks for one of the four possible shadow conditions: both, left, right, neither. Depending on which condition is detected, **PULSOUT** commands deliver pulses for one of the following maneuvers: forward, pivot right, pivot left, or sit still. Regardless of the condition, one of the four sets of pulses will be delivered each time through the **DO…LOOP**. After the **IF…THEN** statement, it's important to remember to include the **PAUSE 20** to ensure the low time between each pair of servo pulses.

Your Turn – Condensing the Program

This program does not need the **ELSE** condition or the two **PULSOUT** commands that follow. If you deliver no pulses, the Boe-Bot will sit still, just as it should when you deliver pulses using 750 for the **PULSOUT** *Duration* argument.

√ Try deleting (or commenting) this code block.

```
ELSE
  PULSOUT 13, 750
  PULSOUT 12, 750
```

√ Run the modified program.

√ Can you detect any difference in the Boe-Bot's behavior?

ACTIVITY #4: GETTING MORE INFORMATION FROM YOUR PHOTORESISTORS

The only information the BASIC Stamp was able to gather from the photoresistor divider circuits was whether the light level was above or below a threshold. This activity introduces a different circuit that the BASIC Stamp can monitor, and actually gather enough information from it to determine relative light levels. The value the BASIC Stamp gets from the circuit will range from small numbers, indicating bright light, to large numbers, indicating low light. This means no more manually replacing series resistors based on light levels. Instead, you will be able to adjust your program to look for different ranges of values.

Introducing the Capacitor

A capacitor is a device that stores charge, and it is a fundamental building block of many circuits. 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 the Boe-Bot. 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's abbreviated μF. The capacitor you will use in this exercise stores one one-hundredth of a millionth of a farad. That's 0.01 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 103 on the 0.01 µF capacitor's case is a measurement picofarads or (pF). 103 is 10, with three zeros added, which is 10,000. Here is how to relate 103 to 0.01 µF.

10,000 is 10×10^3.

$(10 \times 10^3) \times (1 \times 10^{-12})$ F = 10×10^{-9} F

which is also 0.01×10^{-6} F

which is 0.01 µF.□

Figure 6-7 shows the schematic symbol for a 0.01 µF capacitor along with a drawing of the part in your Boe-Bot parts kit. The 103 marking on the capacitor indicates its value.

Parts List:

(2) Photoresistors - CDS
(2) Capacitors – 0.01 µF (103)
(2) Resistors - 220 Ω (red-red-brown)
(2) Jumper wires

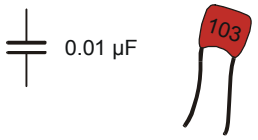


Figure 6-7 Capacitor Schematic Symbol and Part Drawing

There may also be 0.1 µF capacitors marked 104 in your kit. Do not use them in these activities.

√ Make sure you have selected the 0.01 µF capacitors (marked 103) for this activity.

The 0.1 µF capacitors can be used in brightly lit areas, but they interfere with the Boe-Bot's performance in indoor and low lighting activities.

Rebuilding the Photosensitive Eyes

The circuit the BASIC Stamp can use to determine light levels is called a resistor/capacitor (RC) circuit. Figure 6-8 shows schematics of the Boe-Bot's RC light detection circuits and Figure 6-9 shows examples wiring diagrams for the Board of Education and the HomeWork Board.

√ Disconnect power from your board and servos.
√ Build the RC circuits shown in Figure 6-8 using Figure 6-9 as a reference.

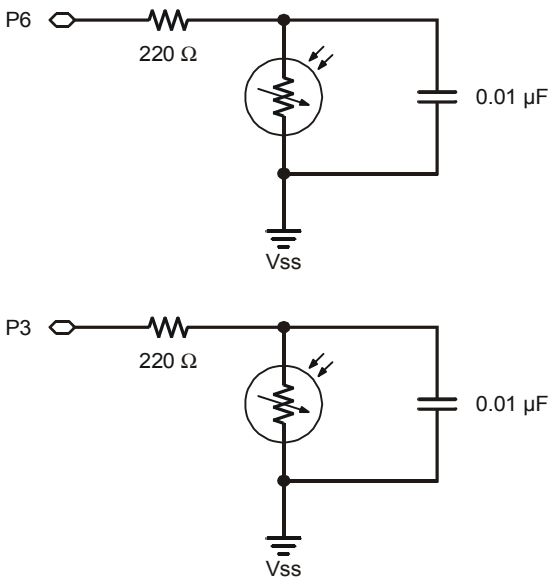


Figure 6-8
Schematic - Two Photoresistor RC Circuits

For measurement of resistance that varies with light.

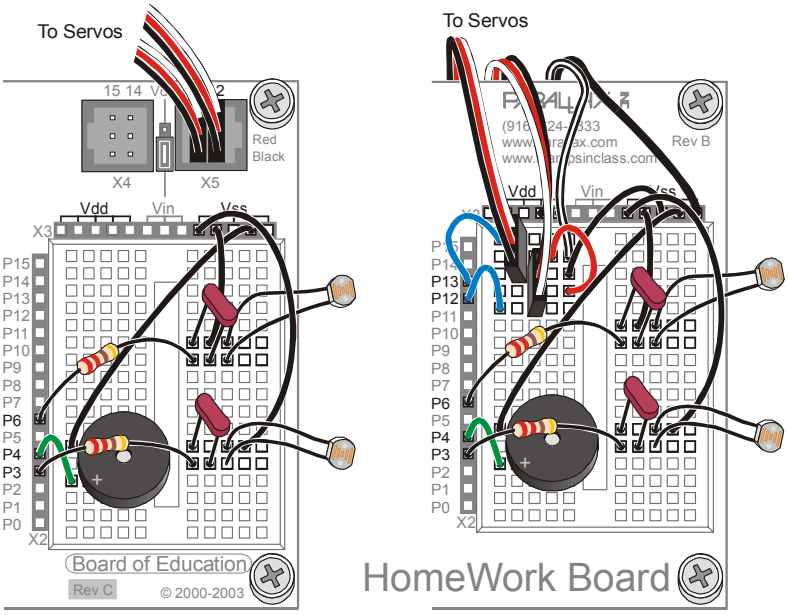


Figure 6-9
Wiring Diagrams for Photoresistor Circuits

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

About RC Decay Time and the Photoresistor Circuit

Think of a capacitor in the circuit shown in Figure 6-10 as a tiny rechargeable battery. When P6 sends the high signal, it essentially charges this capacitor-battery by applying 5 V to it. After a few ms, the capacitor charges up to almost 5 V. If the BASIC Stamp program then changes the I/O pin so that it just quietly listens, the capacitor loses its charge through the photoresistor. As the capacitor looses its charge through the photoresistor, its voltage decays, getting lower and lower as it looses charge. The amount of time it takes for the voltage that **IN6** senses to drop below 1.4 V depends on how strongly the photoresistor "resists" the flow of electric current supplied by the capacitor. If the photoresistor has a large resistance value due to very dim lighting conditions, the capacitor takes longer to discharge. If the photoresistor has a small resistance value because the light incident on its surface is very bright, it will not resist current very strongly, and the capacitor will lose its charge very rapidly.

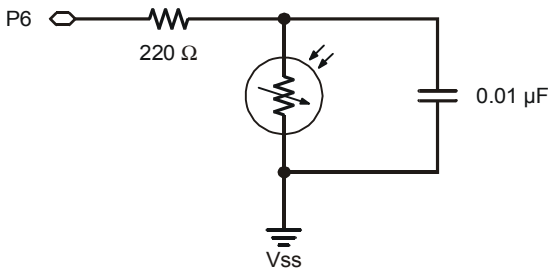


Figure 6-10
RC Circuit
Connected to I/O
Pin

Connected in parallel

The photoresistor and capacitor shown in Figure 6-10 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 photoresistor and the capacitor each have one lead connected to Vss. They also each have one lead connected to the same 220 Ω resistor lead.□

Measuring RC Decay Time with the BASIC Stamp

The BASIC Stamp can be programmed to charge the capacitor and then measure the time it takes the capacitor's voltage to decay to 1.4 V. This decay time measurement can be used to indicate the photoresistor's resistance. This resistance in turn indicates how bright the light detected by the photoresistor really is. This measurement requires a combination of the **HIGH** and **PAUSE** commands along with a new command called

RCTIME. The **RCTIME** command is designed to measure RC decay time on a circuit like the one in Figure 6-10. Here is the syntax for the **RCTIME** command:

RCTIME ***Pin, State, Duration***

The ***Pin*** argument is the number of the I/O pin that you want to measure. For example, if you want to measure P6, the ***Pin*** argument should be 6. The ***State*** argument can either be 1 or 0. It should be 1 if the voltage across the capacitor starts above 1.4 V and decays downward. It should be 0 if the voltage across the capacitor starts below 1.4 V and grows upward. For the circuit in Figure 6-10, the voltage across the capacitor will start close to 5 V and decay to 1.4 V, so the ***State*** argument should be 1. The ***Duration*** argument has to be a variable that stores the time measurement, which is in 2 µs units. In this next example program, we'll measure the RC decay time on the photoresistor circuit connected to P6, which is the photoresistor on the Boe-Bot's left.

To measure RC decay, the first thing you have to do is make sure you have declared a variable that will store the time measurement:

```
timeLeft    VAR   Word
```

These next three lines of code charge the capacitor, measure the RC decay time and then store it in the **timeLeft** variable.

```
HIGH 6
PAUSE 3
RCTIME 6,1,timeLeft
```

To get the measurement, the code implements these three steps:

1. Start charging the capacitor by connecting the circuit to 5 V (using the **HIGH** command).
2. Use **PAUSE** to give the **HIGH** command enough time to charge the capacitor in the RC circuit.
3. Execute the **RCTIME** command, which sets the I/O pin to input, measures the decay time (from almost 5 V to 1.4 V), and stores it in the **timeLeft** variable.

Example Program: TestP6Photoresistor.bs2

√ Reconnect power to your board.
√ Enter, save, and run TestP6Photoresistor.bs2.

- √ Cast a shadow over the photoresistor connected to P6 and verify that the time measurement gets larger as the environment gets darker.
- √ Point the photoresistor's light collecting surface directly at an overhead light, or shine flashlight directly at it. The time measurement should get very small. It should then get larger as you gradually direct the photoresistor further away from the light source. It should get even larger if you cast a shadow over it or turn out the lights.

```
' Robotics with the Boe-Bot - TestP6Photoresistor.bs2
' Test Boe-Bot photoresistor circuit connected to P6 and display
' the decay time.

' {$STAMP BS2}                                ' Stamp directive.
' {$PBASIC 2.5}                               ' PBASIC directive.

timeLeft       VAR     Word

DO

  HIGH 6
  PAUSE 2
  RCTIME 6,1,timeLeft

  DEBUG HOME, "timeLeft = ", DEC5 timeLeft
  PAUSE 100

LOOP
```

Your Turn

- √ Save TestP6Photoresistor.bs2 as TestP3Photoresistor.bs2.
- √ Modify the program so that it performs the RC decay time measurement on the right photoresistor, the one connected to P3.
- √ Repeat the shadow and bright light tests with the P3 RC circuit and verify that it works correctly. You will need to modify the ***Pin*** arguments for both the **HIGH** and **RCTIME** commands, changing them from 6 to 3.

ACTIVITY #5: FLASHLIGHT BEAM FOLLOWING BOE-BOT

In this activity, you will test and calibrate your Boe-Bot's light sensors so that they recognize the difference between ambient light and a directed flashlight beam. You will then program the Boe-Bot to follow the flashlight beam that is pointed at the surface in front of the Boe-Bot.

Extra Equipment

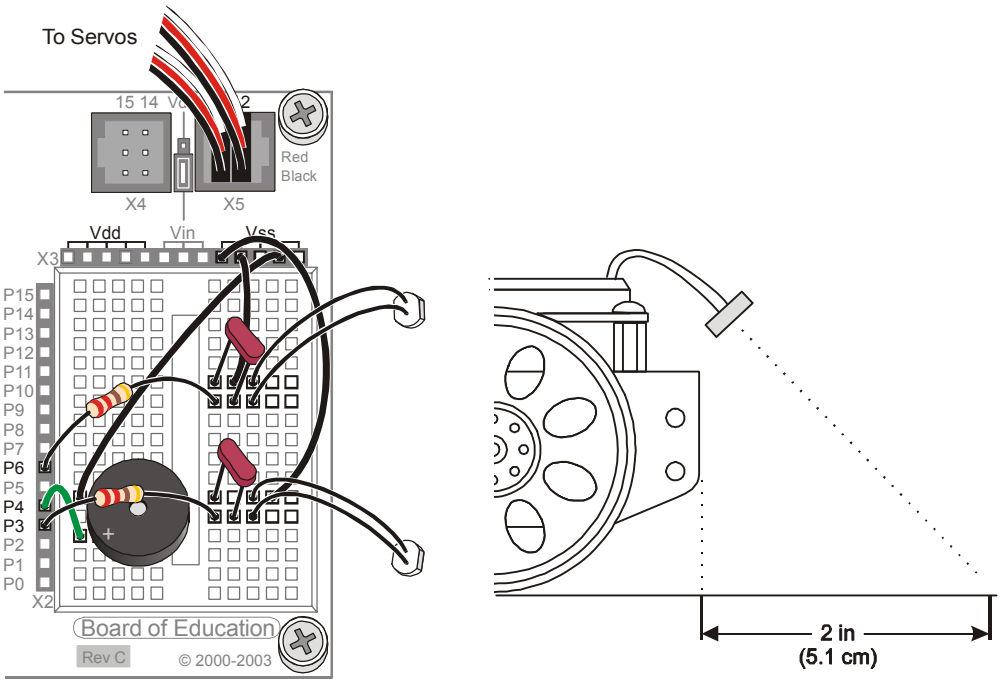
(1) Flashlight

Adjust Sensors to Search for Flashlight Beam

This activity works best if the photoresistors' light-collecting surfaces are pointing ahead at separate points on the ground about 2 in (5.1 cm) in front of the Boe-Bot.

√ Point the light collecting surfaces of your photoresistors at the surface in front of the Boe-Bot as shown in Figure 6-11.

Figure 6-11: Photoresistor Orientation



Testing Sensor Response to Flashlight Beam

Before you can program the Boe-Bot to turn towards a flashlight beam, you have to know the difference between light readings with and without the flashlight beam shining in the Boe-Bot's path.

Example Program: TestBothPhotoresistors.bs2

- √ Enter, save, and run TestBothPhotoresistors.bs2.
- √ Place the Boe-Bot on the surface where it is to follow the flashlight beam. Make sure it is still connected to the serial cable and that the measurements are displaying in the Debug Terminal.
- √ Record the values of both time measurements in the first row of Table 6-1.
- √ Turn on your flashlight, and focus your beam in front of the Boe-Bot.
- √ Your time measurements should now be significantly lower than the first set. Record these new values of both time measurements in the second row of Table 6-1.

Table 6-1: RC-Time Measurements With and Without Flashlight Beam

Duration Values		Description
timeLeft	**timeRight**	
		Time measurements with no flashlight beam (ambient light).
		Time measurements with flashlight beam focused in front of the Boe-Bot.

```
' Robotics with the Boe-Bot - TestBothPhotoresistors.bs2
' Test Boe-Bot RC photoresistor circuits.

' {$STAMP BS2}                                 ' Stamp directive.
' {$PBASIC 2.5}                                ' PBASIC directive.

timeLeft      VAR     Word                    ' Variable declarations.
timeRight     VAR     Word

DEBUG "PHOTORESISTOR VALUES", CR,             ' Initialization.
      "timeLeft  timeRight", CR,
      "--------  ---------"

DO                                            ' Main routine.

  HIGH 6                                      ' Left RC time measurement.
```

```
  PAUSE 3
  RCTIME 6,1,timeLeft

  HIGH 3                                        ' Right RC time measurement.
  PAUSE 3
  RCTIME 3,1,timeRight

  DEBUG CRSRXY, 0, 3,                           ' Display measurements.
        DEC5 timeLeft,
        "     ",
        DEC5 timeRight

  PAUSE 100

LOOP
```


Your Turn

- √ Try facing the Boe-Bot in different directions, and repeat your measurements.
- √ For better results, you can average your measurements for "flashlight on" and "flashlight off" and replace the values in Table 6-1 with your average values.

Following the Flashlight Beam

You have been using variable declarations up to this point. For example, **`counter VAR Nib`** gives the name **`counter`** to a particular memory location in the BASIC Stamp's RAM. After you have declared the variable, every time you use **`counter`** in a PBASIC program, it uses the value stored at that particular location in the BASIC Stamp's RAM.

You can also declare constants. In other words, if you have a number you plan on using in your program, give it a useful name. Instead of the **`VAR`** directive, use the **`CON`** directive. Here are some **`CON`** directives from the next example program:

```
LeftAmbient     CON     108
RightAmbient    CON     114
LeftBright      CON     20
RightBright     CON     22
```

Now, everywhere in the program the name **`LeftAmbient`** is used, the BASIC Stamp will use the number 108. Everywhere **`RightAmbient`** is used, the BASIC Stamp will use the value 114. Likewise, everywhere **`LeftBright`** appears, it's really the value 20, and **`RightBright`** is 22. You will substitute your values from Table 6-1 before running the program.

Constants can even be used to calculate other constants. Here is an example of two constants, named **LeftThreshold** and **RightThreshold** that are calculated using the four constants just discussed. The **LeftThreshold** and **RightThreshold** constants are used in the program to figure out whether or not the flashlight beam has been detected.

```
'                        Average                        Scale factor

LeftThreshold  CON     LeftBright + LeftAmbient / 2    * 5 / 8
RightThreshold CON     RightBright + RightAmbient / 2 * 5 / 8
```

The math performed on these constants is an average, and then a scale. The average calculation for **LeftThreshold** is **LeftBright** + **LeftAmbient** / 2. That result is multiplied by 5 and divided by 8. This means that **LeftThreshold** is a constant whose value is the $^{5}/_{8}$ of the average of **LeftBright** and **LeftAmbient**.

Math expressions in PBASIC are executed from left to right. First, **LeftBright** is added to **LeftAmbient**. This value is divided by 2. The result is then multiplied by 5 and divided by 8.

Let's try this: **LeftBright** + **LeftAmbient** = 20 + 108 = 128.

128 / 2 = 64.

64 * 5 = 320

320 / 8 = 40

You can use parentheses to force a calculation that is further to the right in a line of PBASIC code to be completed first. For 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.

Example Program: FlashlightControlledBoeBot.bs2

√ Enter FlashlightControlledBoeBot.bs2 into the BASIC Stamp Editor.
√ Substitute your **timeLeft** measurement with no flashlight beam (from Table 6-1) in place of the value 108 in the **LeftAmbient CON** directive.

- √ Substitute your **timeRight** measurement with no flashlight beam in place of the value 114 in the **RightAmbient CON** directive.
- √ Substitute your **timeLeft** measurement with focused flashlight beam in place of the value 20 in the **LeftBright CON** directive.
- √ Substitute your **timeRight** measurement with focused flashlight beam in place of the value 22 in the **RightBright CON** directive.
- √ Reconnect power to your board and servos.
- √ Save and then run FlashlightControlledBoeBot.bs2.
- √ Experiment and figure out exactly where to focus the light to get the forward, left turn, and right turn maneuvers to execute.
- √ Use the flashlight to guide your Boe-Bot through various obstacle courses and maneuvers.


```
' -----[ Title ]-----------------------------------------------------------
' Robotics with the Boe-Bot - FlashlightControlledBoeBot.bs2
' Boe-Bot follows flashlight beam focused in front of it.

' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Constants ]-------------------------------------------------------

' REPLACE THESE VALUES WITH THE VALUES YOU DETERMINED AND ENTERED INTO
' TABLE 6.1.

LeftAmbient    CON     108
RightAmbient   CON     114
LeftBright     CON     20
RightBright    CON     22

'                       Average                       Scale factor

LeftThreshold  CON     LeftBright + LeftAmbient / 2   * 5 / 8
RightThreshold CON     RightBright + RightAmbient / 2 * 5 / 8

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

' Declare variables for storing measured RC times of the
' left & right photoresistors.

timeLeft       VAR     Word
timeRight      VAR     Word

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

```
FREQOUT 4, 2000, 3000

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

DO

  GOSUB Test_Photoresistors
  GOSUB Navigate

LOOP

' -----[ Subroutine - Test_Photoresistors ]----------------------------------

Test_Photoresistors:

  HIGH 6                                    ' Left RC time measurement.
  PAUSE 3
  RCTIME 6,1,timeLeft

  HIGH 3                                    ' Right RC time measurement.
  PAUSE 3
  RCTIME 3,1,timeRight

  RETURN

' -----[ Subroutine - Navigate ]--------------------------------------------

Navigate:

  IF (timeLeft < LeftThreshold) AND (timeRight < RightThreshold) THEN
    PULSOUT 13, 850                         ' Both detect flashlight beam,
    PULSOUT 12, 650                         ' full speed forward.
  ELSEIF (timeLeft < LeftThreshold) THEN    ' Left detects flashlight beam,
    PULSOUT 13, 700                         ' pivot left.
    PULSOUT 12, 700
  ELSEIF (timeRight < RightThreshold) THEN  ' Right detects flashlight beam,
    PULSOUT 13, 800                         ' pivot right.
    PULSOUT 12, 800
  ELSE
    PULSOUT 13, 750                         ' No flashlight beam, sit still.
    PULSOUT 12, 750
  ENDIF

  PAUSE 20                                  ' Pause between pulses.

  RETURN
```

How FlashlightControlledBoeBot.bs2 Works

These are the four constant declarations that you used with your own values from Table 6-1.

```
LeftAmbient     CON     108
RightAmbient    CON     114
LeftBright      CON     20
RightBright     CON     22
```

Now that the four constants have been declared, the next two lines average and scale the values to come up with threshold values for the program. These threshold values can be compared with the current **`timeLeft`** and **`timeRight`** measurements to determine whether the photoresistors are sensing ambient light or a focused beam.


```
'                       Average                         Scale

LeftThreshold  CON     LeftBright + LeftAmbient / 2    * 5 / 8
RightThreshold CON     RightBright + RightAmbient / 2 * 5 / 8
```

These variables are used to store the **`RCTIME`** measurements.

```
timeLeft        VAR     Word
timeRight       VAR     Word
```

This is the reset indicator that has been used in most of the programs in this text.

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

The Main Routine section contains just two subroutine calls. All the actual work in the program occurs in the two subroutines. **`Test_Photoresistors`** takes the **`RCTIME`** measurements for both RC photoresistor circuits, and the **`Navigate`** subroutine makes the decisions and delivers the servo pulses.

```
DO

  GOSUB Test_Photoresistors
  GOSUB Navigate

LOOP
```

This is the subroutine that performs the **RCTIME** measurements on both photoresistor RC circuits. The measurement for the left circuit is stored in the **timeLeft** variable, and the measurement for the right circuit is stored in the **timeRight** variable.

```
Test_Photoresistors:

  HIGH 6
  PAUSE 3
  RCTIME 6,1,timeLeft

  HIGH 3
  PAUSE 3
  RCTIME 3,1,timeRight

  RETURN
```

The Navigate subroutine uses an **IF…THEN** statement to compare the **timeLeft** variable against the **LeftThreshold** constant and the **timeRight** variable against the **RightThreshold** constant. Remember, when the **RCTIME** measurement is small, it means bright light is detected, and when it's large, it means the light is not as bright. So, when one of the variables that stores an **RCTIME** measurement is smaller than the threshold constant, it means the flashlight beam has been detected; otherwise, the flashlight beam has not been detected. Depending on which condition this subroutine detects (both, left, right or neither), the correct navigation pulses is applied, followed by a **PAUSE** before the **RETURN** command exits the subroutine.

```
Navigate:

  IF(timeLeft<LeftThreshold)AND(timeRight<RightThreshold) THEN
    PULSOUT 13, 850
    PULSOUT 12, 650
  ELSEIF (timeLeft < LeftThreshold) THEN
    PULSOUT 13, 700
    PULSOUT 12, 700
  ELSEIF (timeRight < RightThreshold) THEN
    PULSOUT 13, 800
    PULSOUT 12, 800
  ELSE
    PULSOUT 13, 750
    PULSOUT 12, 750
  ENDIF
```

```
        PAUSE 20

        RETURN
```

Your Turn – Adjusting the Performance and Changing the Behavior

You can adjust the program's performance by adjusting the scale factor term in this constant declaration:

```
'                       Average                         Scale factor

LeftThreshold  CON     LeftBright + LeftAmbient / 2   * 5 / 8
RightThreshold CON     RightBright + RightAmbient / 2 * 5 / 8
```


If you change the scale factor from $^5/_8$ to $^1/_2$, it will make the Boe-Bot less sensitive to the flashlight, which may (or may not) lead to improved flashlight control.

- √ Try different scale factors, such as $^1/_4$, $^1/_2$, $^1/_3$, $^2/_3$, and $^3/_4$ and make notes about any differences in the way the Boe-Bot responded to the flashlight beam.

By modifying the **IF…THEN** statement in the example program, you can change the Boe-Bot's behavior so that it tries to get the light out of its eyes.

- √ Modify the **IF…THEN** statement so that the Boe-Bot backs up when it detects the flashlight beam with both photoresistor circuits and turns away if it detects the flashlight beam with only one of its photoresistor circuits.

ACTIVITY #6: ROAMING TOWARD THE LIGHT

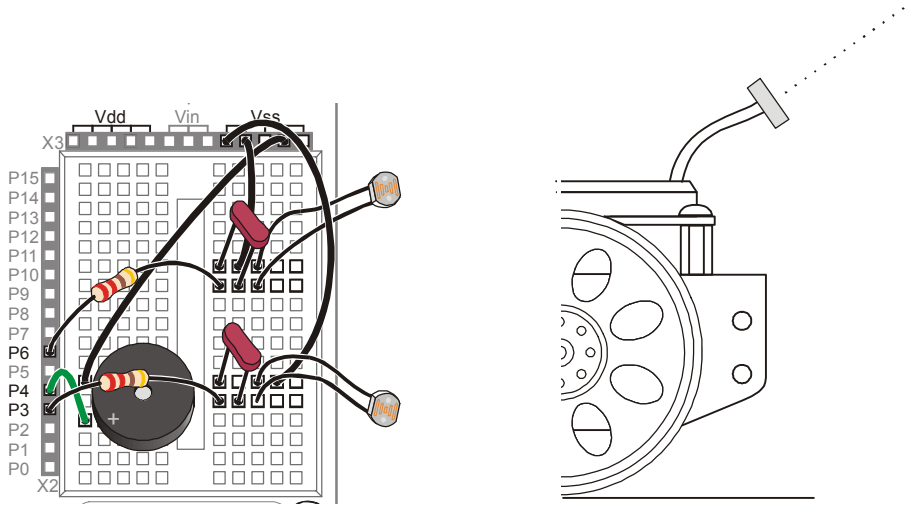
The example program in this activity can be used to guide the Boe-Bot through exiting a fairly dark room toward a doorway that's letting in brighter light. It also allows for much better control over the Boe-Bot's roaming by casting shadows over the photoresistors with your hand.

Readjusting the Photoresistors

This activity works best if the photoresistors' light collecting surfaces are pointing upwards and outwards.

√ Point the light collecting surfaces of your photoresistors upward and outward shown in Figure 6-12.

Figure 6-12: Photoresistor Orientation



Programming the Roaming Toward the Light Behavior

The key to roaming toward brighter light sources is going straight ahead when the differences between the photoresistor measurements are small, and turning toward the smaller photoresistor measurement when there is a large difference between the two measurements. In effect, this means the Boe-Bot will turn toward bright light.

Initially this seems like a simple enough programming task; **`IF…THEN`** reasoning like this example below should work. The problem is, it doesn't because the Boe-Bot gets stuck turning left and then right again because the change in **`timeLeft`** and **`timeRight`** is too large. Each time the Boe-Bot turns a little, the **`timeRight`** and **`timeLeft`** variables change so much that the Boe-Bot tries to correct and turn back. It never manages to get any forward pulses in.

```
IF (timeLeft > timeRight) THEN              ' Turn right.
  PULSOUT 13, 850
  PULSOUT 12, 850
ELSEIF (timeRight > timeLeft) THEN          ' Turn left.
```

```
  PULSOUT 13, 650
  PULSOUT 12, 650
ELSE                                          ' Go forward.
  PULSOUT 13, 850
  PULSOUT 12, 650
ENDIF
```

Here is another code block that works a little better. This code block fixes the turning back and forth problem under certain conditions. The **`timeLeft`** variable now has to be larger than **`timeRight`** by a margin of 15 before the Boe-Bot will apply a left pulse. Likewise, **`timeRight`** has to be larger than **`timeLeft`** by 15 before the Boe-Bot adjusts to the left. This gives the Boe-Bot the opportunity to apply enough forward pulses before it has to correct with a turn, but only at certain light levels.


```
IF (timeLeft > timeRight + 15) THEN        ' Turn right.
  PULSOUT 13, 850
  PULSOUT 12, 850
ELSEIF (timeRight > timeLeft + 15) THEN    ' Turn left.
  PULSOUT 13, 650
  PULSOUT 12, 650
ELSE                                       ' Go forward.
  PULSOUT 13, 850
  PULSOUT 12, 650
ENDIF
```

The problem with the code block above is that it works under medium dark conditions only. If you take it into a much darker area, the Boe-Bot starts turning back and forth again, and it never applies any forward pulses. If you take it into a brighter area, the Boe-Bot just goes forward, and never makes any adjustments to the left or right.

Why does that happen?

Here is the answer: When the Boe-Bot is in a dark part of a room, the measurement for each photoresistor will be large. For the Boe-Bot to decide to turn toward a light source, the difference between these two measurements has to be large. When the Boe-Bot is in a more brightly lit area, the measurement for each photoresistor will be smaller. For the Boe-Bot to decide to make a turn, the difference between photoresistor measurements also has to be much smaller than it was in the darker part of the room. The way to make this difference respond to the lighting conditions is to make it a variable that is a fraction of the average of **`timeRight`** and **`timeLeft`**. That way, it will always be the right value, regardless whether the lighting is bright or dim.

```
  average = timeRight + timeLeft / 2
  difference = average / 6
```

Now, the **difference** variable can be used in this **IF…THEN** statement, and it will be a large value when the lighting is low, and a small value when the lighting is bright.

```
IF (timeLeft > timeRight + difference) THEN  ' Turn right.
  PULSOUT 13, 850
  PULSOUT 12, 850
ELSEIF (timeRight > timeLeft + difference) THEN ' Turn left.
  PULSOUT 13, 650
  PULSOUT 12, 650
ELSE                                          ' Go forward.
  PULSOUT 13, 850
  PULSOUT 12, 650
ENDIF
```

Example Program – RoamingTowardTheLight.bs2

Unlike RoamingWithPhotoresistorDividers.bs2 on page 201, this program will be very responsive to your hand casting a shadow over the photoresistor, regardless of whether the light is bright or dim. This program does not need to change resistors depending on the lighting conditions. Instead, it takes into account the lighting conditions and the sensitivity adjustment is made in software using the **average** and **difference** variables.

For this program to work well, your photoresistors should respond similarly to similar light levels. If the RC circuits are severely mismatched, your measurements from Table 6-1 will be very different under the same lighting conditions. You can correct these mismatched measurements using techniques discussed in Appendix F: Balancing Photoresistors.

This program measures the overall average of **timeLeft** and **timeRight** and uses it to set the **difference** between the **timeLeft** and **timeRight** measurements that's needed to justify delivering a turning pulse.

- √ Enter, save, and run RoamingTowardTheLight.bs2
- √ Take it to various areas, and let it roam, and verify that you can change its course by casting a shadow over one of the photoresistor RC circuits, regardless of the lighting conditions.

√ Also try placing your Boe-Bot in a room that is poorly lit, but that has light streaming in through a doorway from an adjacent brightly lit room or hallway. See if the Boe-Bot can successfully find its way out the door.

```
' -----[ Title ]-----------------------------------------------------------
' Robotics with the Boe-Bot - RoamingTowardTheLight.bs2
' Boe-Bot roams, and turns away from dark areas in favor of brighter areas.

' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

DEBUG "Program Running!"

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

' Declare variables for storing measured RC times of the
' left & right photoresistors.

timeLeft       VAR     Word
timeRight      VAR     Word
average        VAR     Word
difference     VAR     Word

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

FREQOUT 4, 2000, 3000

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

DO
  GOSUB Test_Photoresistors
' For mismatched photoresistors, use Appendix F, uncomment and use next line.
' timeLeft = (timeLeft */ 351) + 7  ' Replace 351 and 7 with your own values.
  GOSUB Average_And_Difference
  GOSUB Navigate
LOOP

' -----[ Subroutine - Test_Photoresistors ]---------------------------------

Test_Photoresistors:

  HIGH 6                                    ' Left RC time measurement.
  PAUSE 3
  RCTIME 6,1,timeLeft

  HIGH 3                                    ' Right RC time measurement.
  PAUSE 3
  RCTIME 3,1,timeRight

  RETURN
```

```
' -----[ Subroutine - Average_And_Difference ]-------------------------------

Average_And_Difference:

  average = timeRight + timeLeft / 2
  difference = average / 6

  RETURN

' -----[ Subroutine - Navigate ]-----------------------------------------------

Navigate:

  ' Shadow significantly stronger on left detector, turn right.
  IF (timeLeft > timeRight + difference) THEN
    PULSOUT 13, 850
    PULSOUT 12, 850
  ' Shadow significantly stronger on right detector, turn left.
  ELSEIF (timeRight > timeLeft + difference) THEN
    PULSOUT 13, 650
    PULSOUT 12, 650
  ' Shadows in same neighborhood of intensity on both detectors.
  ELSE
    PULSOUT 13, 850
    PULSOUT 12, 650
  ENDIF

  PAUSE 10

  RETURN
```

Why `PAUSE 10` instead of `PAUSE 20`? Because the **`Test_Photoresistors`** subroutine has two **`PAUSE`** commands adding up to 6 ms plus some extra time to execute the **`RCTIME`** commands. Both these factors add to the amount of time between servo pulses, so the **`PAUSE`** in the **`Navigate`** subroutine has to be reduced. After some trial and error experiments, **`PAUSE 10`** appeared to give the servos the most reliable performance over the widest range of light levels.

Your Turn – Adjusting the Sensitivity to Differences in Light

Right now, the **`difference`** variable is the **`average`** divided by 6. You can divide **`average`** by a smaller value if you want to make the Boe-Bot less sensitive to differences in light or divide it by a larger value if you want to make the Boe-Bot more sensitive to differences in light level.

- √ Instead of the value 6, try dividing the **average** variable by the values 3, 4, 5, 7, and 9.
- √ Run the program and test the Boe-Bot's ability to exit a dark room with each denominator value.
- √ Decide what the optimum denominator value is.

```
Average_And_Difference:

  average = timeRight + timeLeft / 2
  difference = average / 6

  RETURN
```


You can also change the denominator into a constant like this:

```
Denominator CON 6
```

Then, in your Average_And_Difference subroutine, you can replace 6 (or the optimum value that you determined) with the **Denominator** constant, like this:

```
Average_And_Difference:

  average = timeRight + timeLeft / 2
  difference = average / Denominator

  RETURN
```

- √ Make the changes just discussed, and verify that the program still works correctly.

You can also use one less variable in this program. Notice that the only time the **average** variable is used is to temporarily hold the average value, then it gets divided by **Denominator** and stored in the **difference** variable. The **difference** variable is needed later, but the **average** variable is not. One way to fix this problem would be to simply use the **difference** variable in place of the **average** variable. It will work fine, and you would no longer need the **average** variable. Here is how the subroutine would look:

```
Average_And_Difference:

  difference = timeRight + timeLeft / 2
```

```
  difference = difference / Denominator

  RETURN
```

There is a better way though.

√ Leave the **Average_And_Difference** routine like this:

```
Average_And_Difference:

  average = timeRight + timeLeft / 2
  difference = average / Denominator

  RETURN
```

√ Next, make this change in the variable declarations:

Figure 6-13: Modify RoamingTowardTheLight.bs2 to Save a Word of RAM

```
' Unchanged code                    ' Changed to save Word of RAM

average        VAR     Word         average        VAR     Word
difference     VAR     Word         difference     VAR     average
```

We don't really need the **average** variable, but the program will make more sense to someone trying to understand it if we use the word **average** in the first line and the word **difference** in the second line. Here is how to create an alias name **difference** for the **average** variable.

```
difference     VAR     average
```

Now, both **average** and **difference** refer to the same word of RAM.

√ Test your modified program and make sure it still works properly.

SUMMARY

This chapter focused on measuring differences in light intensity and programming the Boe-Bot to act on these differences. A pair of cadmium sulfide (CdS) photoresistors were used to measure differences in visible light. The CdS photoresistors were first connected to resistors to form voltage dividers, and the BASIC Stamp monitored the voltage at the connection between the photoresistor and the fixed resistor. When this voltage dropped below or rose above 1.4 V the input register for the I/O pin connected to the circuit stored either a 0 or 1. The Boe-Bot was programmed to make decisions using these binary values in a manner similar to the whiskers.

The photoresistor divider technique works so long as the right resistors are chosen and the lighting doesn't change. However, a much more versatile way of detecting light levels with the BASIC Stamp is to use the CdS photoresistor in an RC circuit, charge the capacitor, and then measure the decay time. RC stands for resistor capacitor, and the capacitor was introduced in this chapter along with a circuit that makes it possible for the BASIC Stamp to measure RC decay time. This is easily done with the BASIC Stamp using the RCTIME command, which is specifically designed for measuring RC decay and growth times.

Constants were introduced as a way to substitute meaningful names for numbers that are used in a PBASIC program. Scaling and averaging were also introduced. Scaling was used to set a threshold value to indicate whether or not a flashlight beam was detected. It was also used to determine the average value of the light levels in an area based on the two photoresistor RC time measurements. This was used to create a threshold that automatically self-adjusted to the overall lighting conditions, eliminating the need to change resistors when the light levels change.

Watch the Boe-Bot in Action at www.parallax.com!

You can see the Boe-Bot solving Chapter 6 Projects 1 and 2 along with other Robotics video clips in the Robo Video Gallery under the Robotics Menu at www.parallax.com.

Questions

1. How does the resistance of a photoresistor respond to bright and dim light? What happens if the light levels are between bright and dim?

2. Does an I/O pin have any effect on the circuit when it's set to input? What causes the input register for an I/O pin to hold a 1 or 0 when it's set to input?
3. What does threshold voltage mean? What's the threshold voltage of a BASIC Stamp I/O pin?
4. Referring to Figure 6-4 on page 197, what causes Vo to rise above or fall below a BASIC Stamp I/O pin's threshold voltage? What is it about the circuit that causes Vo to change value?
5. How does the program ShadowGuidedBoeBot.bs2 differ from the program RoamingWithPhotoresistorDividers.bs2? What does this change in the Boe-Bot's performance?
6. What is a constant declaration? What does it do? How can you use one in a program?
7. How are math expressions evaluated in PBASIC?
8. What are the two examples in this chapter where PBASIC was used to calculate an average? How are they different? How are they the same?

Exercises

1. Calculate Vo for Figure 6-4 on page 197 if R is 10 kΩ. Repeat for R = 30 kΩ.
2. If Vo in Figure 6-4 on page 197 is 1.4 V, what's the value of R? Repeat for Vo = 1 V and Vo = 3 V.
3. Assume you have three variable values: **`firstValue`**, **`secondValue`**, and **`thirdValue`**. Write a command that takes the average of these three values in a variable named **`myAverage`**. Write a command that stores 7/8 of the average value in a variable named **`myScaledAverage`**. Write the variable declarations needed to make your command able to run in a program, first with **`myAverage`** and **`myScaledAverage`** as separate variables, then with one of these variable names aliased as the other.

Projects

1. With your Boe-Bot's photoresistors looking down in front of it, develop a program that makes your Boe-Bot recognize the difference between black and white. Find a large white surface and place dark-black sheets of paper on it. Develop a program that makes the Boe-Bot avoid the black sheets of paper. Hints: Make sure to test and understand what the Boe-Bot sees when it is focused on a black sheet of paper and what it sees when it is focused on a white background. Use example programs from the last three activities in this chapter. The RC decay time circuit and programs will be much more helpful for making

the program work than the photoresistor divider techniques. Also, make sure this obstacle course is in a uniformly lit area. Bright sunlight from windows, and shadows cast by onlookers can make the demonstration fail.

2. If you succeeded with project 1, experiment with confining the Boe-Bot so that it can only roam in a space that is enclosed by black sheets of paper.

Solutions

Q1. The resistance is small, a few ohms, if the light is bright. The resistance is large, around 50 kΩ, in dim light. For light levels between bright and dim, its resistance will be somewhere between the bright and dim values.

Q2. No. The I/O pin just quietly listens without any actual effect on the circuit. The value of the applied voltage causes the input register to change what it stores. If the applied voltage is less than 1.4 volts it stores a 0. Otherwise it stores a 1.

Q3. The threshold voltage is a value above which is a logic 1, below which is a logic 0. The threshold voltage of BASIC Stamp modules is 1.4 volts.

Q4. The value of Vo is determined by the ratio of the resistors. Vo changes value because the resistor, R, changes value. R is a photoresistor which changes value depending on the amount of light falling upon it.

Q5. It checks the sensors between each pulse, instead of having fixed maneuvers of many pulses. This makes the Boe-Bot much more responsive.

Q6. A constant declaration tells the compiler the value of your constant. For example, MaxTemp `CON 212` is a constant declaration. A constant gives a useful name to a number or value used in a program. To use a constant, type the constant name anywhere in the program where the value is needed.

Q7. Expressions are evaluated from left to right. This is different than standard algebraic evaluation, where multiplication and division are evaluated before addition and subtraction.

Q8. In FlashLightControlledBoeBot.bs2, averages were used to calculate lighting thresholds. In RoamingTowardTheLight.bs2, an average of the left and right readings is calculated. They different in that the first is calculated statically in a constant declaration, and the second is calculated dynamically as the program runs. They are similar in that they both add two values together and divide by 2.

E1.

a) **R = 10 kOhm**

Vo = 5V * (2000 / (2000 + R))
= 5 * (2000 / (2000 + 10000)
= 5 * (2000 / (12000)
= 5 * (2 / 12)
= 5 * (1 / 6)
= 5 * 0.17
= 0.83 Volts
If R = 10 kOhm, Vo = 0.83 V

b) **R = 30 kOhm**

Vo = 5V * (2000 / (2000 + R))
= 5 * (2000 / (2000 + 30000)
= 5 * (2000 / (32000)
= 5 * (2 / 32)
= 5 8 (1 / 16)
= 5 * 0.06
= 0.31 Volts
If R = 30 kOhm, Vo = 0.31 V

E2.

a) Vo = 1.4 V

R = (5 * (2000/Vo)) – 2000
= (5 * (2000/1.4)) – 2000
= (5 * 1428.57) – 2000
= 7142.86 – 2000
= 5142.86
= 5143 Ohm
When Vo = 1.4V, R = 5143 Ω

b) Vo = 1.0 V

R = (5 * (2000/1)) – 2000
= (5 * 2000) – 2000
= (10000) – 2000
= 8000
= 8 kOhm
When Vo = 1.4V, R = 8 kΩ

c) Vo = 3.0 V

R = (5 * (2000/3.0)) – 2000
= (5 * 666.67) – 2000
= 3333.33 – 2000
= 1333.33
= 1333 Ohm
When Vo = 1.4V, R = 1333 Ω

E3. The average of these three values in a variable named **`myAverage`**, storing 7/8 in **`myScaledAvereage`**:

```
myAverage = firstValue + secondValue + thirdValue / 3
myScaledAverage = myAverage * 7 / 8
```

Declarations as separate variables:

```
myAverage       VAR     Word
myScaledAverage VAR     Word
```

Declarations using aliasing:

```
myAverage       VAR     Word
myScaledAverage VAR     myAverage
```

P1. The first step is to use "TestBothPhotoresistors.bs2" and determine the values for the white surface and the black paper. Similar to "FlashlightControlledBoeBot.bs2", these values can be coded as constants. Then, **`IF…THEN`** statements can be used to determine whether the values are above or below the average readings. (For the author's Boe-Bot, scaling was not necessary). Here's a program solution that makes the Boe-Bot recognize the difference between black and white surfaces.

```
' -----[ Title ]-----------------------------------------------------------
' Robotics with the Boe-Bot - TestBlackWhiteLogic.bs2
' Calculate whether Boe-Bot is over black or white surface, and print.

' {$STAMP BS2}                              ' Stamp directive
' {$PBASIC 2.5}                             ' PBASIC directive.

' -----[ Constants ]-------------------------------------------------------

LeftWhite      CON     16
RightWhite     CON     33
LeftBlack      CON     26
RightBlack     CON     45
LeftAvg        CON     LeftWhite + LeftBlack / 2
RightAvg       CON     RightWhite + RightBlack / 2

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

timeLeft       VAR     Word                ' Left photoresistor reading
timeRight      VAR     Word                ' Right photoresistor reading

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

DO
  GOSUB Test_Photoresistors
  IF (timeLeft > LeftAvg) THEN
    DEBUG CRSRXY, 0, 0, "Left Black  "
  ELSE
    DEBUG CRSRXY, 0, 0, "Left White  "
  ENDIF
  IF (timeRight > RightAvg) THEN
    DEBUG CRSRXY, 13, 0, "Right Black", CR
  ELSE
    DEBUG CRSRXY, 13, 0, "Right White", CR
  ENDIF
LOOP

' -----[ Subroutine - Test_Photoresistors ]-------------------------------

Test_Photoresistors:
  HIGH 6                                    ' Left RC time Measurement.
  PAUSE 3
  RCTIME 6,1,timeLeft
  HIGH 3                                    ' Right RC time measurement.
  PAUSE 3
  RCTIME 3,1,timeRight
RETURN
```

To develop a program that makes the Boe-Bot avoid the black sheets of paper, the decision and navigation steps required are very similar to

"FlashlightControlledBoeBot.bs2" and "RoamingTowardTheLight.bs2". A sample solution is shown below.

```
' -----[ Title ]-----------------------------------------------------
' Robotics with the Boe-Bot - AvoidBlackSpots.bs2
' Boe-Bot avoids black pieces of paper.

' {$STAMP BS2}                              ' Stamp directive
' {$PBASIC 2.5}                             ' PBASIC directive.

' -----[ Constants ]-------------------------------------------------

LeftWhite      CON     16
RightWhite     CON     33
LeftBlack      CON     26
RightBlack     CON     45
LeftAvg        CON     LeftWhite + LeftBlack / 2
RightAvg       CON     RightWhite + RightBlack / 2

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

timeLeft       VAR     Word                ' Left photoresistor
reading
timeRight      VAR     Word                ' Right photoresistor
reading

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

FREQOUT 4, 2000, 3000

' -----[ Main Routine ]----------------------------------------------
DO
  GOSUB Test_Photoresistors
  GOSUB Navigate
LOOP

' -----[ Subroutines -------------------------------------------------

Test_Photoresistors:
  HIGH 6                                   ' Left RC time
Measurement.
  PAUSE 3
  RCTIME 6,1,timeLeft
  HIGH 3                                   ' Right RC time
measurement.
  PAUSE 3
  RCTIME 3,1,timeRight
RETURN

Navigate:
```

```
' Both detect black paper, back up and make a noise
  IF (timeLeft > LeftAvg) AND (timeRight > RightAvg) THEN
    PULSOUT 13, 650
    PULSOUT 12, 850
    FREQOUT 4, 20, 4400                      ' Beep instead of pause
' Left detects black paper, turn away to right, make a noise
  ELSEIF (timeLeft > LeftAvg) THEN
    PULSOUT 13, 850
    PULSOUT 12, 850
    FREQOUT 4, 20, 2200
' Right detects black paper, turn away to left, make a noise
  ELSEIF (timeRight > RightAvg) THEN
    PULSOUT 13, 650
    PULSOUT 12, 650
    FREQOUT 4, 20, 3300
' Neither detects black paper, go forward one pulse.
  ELSE
   PULSOUT 13,850
   PULSOUT 12,650
   PAUSE 20
  ENDIF
RETURN
```

Hints: Make sure to test and understand what the Boe-Bot sees when it is focused on a black sheet of paper and what it sees when it is focused on a white background. Use example programs from the last three activities in this chapter. The RC decay time circuit and programs will be much more helpful for making the program work than the photoresistor divider techniques. Also, make sure this obstacle course is in a uniformly lit area. Bright sunlight from windows, and shadows cast by onlookers can make the demonstration fail.

P2. The "AvoidBlackSpots.bs2" program solution, above, works quite well to keep the Boe-Bot confined in a black-bordered space. A video clip of a Boe-Bot doing just this can be viewed at www.parallax.com. Under the Robotics menu, look for Robo Video Gallery.

Chapter 7: Navigating with Infrared Headlights

Today's hottest products seem to have one thing in common: wireless communication. Personal organizers beam data into desktop computers, and wireless remotes let us channel surf. Many remote controls and PDA's use signals in the infrared frequency range to communicate, below the visible light spectrum. With a few inexpensive and widely available parts, the BASIC Stamp can also receive and transmit infrared light signals.

USING INFRARED HEADLIGHTS TO SEE THE ROAD

Detecting objects without whiskers doesn't require anything as sophisticated as machine vision. Some robots use RADAR or SONAR (sometimes called SODAR when used in air instead of water). An even simpler system is to use infrared light to illuminate the robot's path and determine when the light reflects off an object. Thanks to the proliferation of infrared (IR) remote controls, IR illuminators and detectors are readily available and inexpensive.

Infrared: Infra means below, so Infra-red is light (or electromagnetic radiation) that has lower frequency, or longer wavelength than red light. Table 7-1 shows the wavelengths for common colors along with the infrared spectrum. Our IR LED and detector work at 980 nm (nanometers) which is considered near infrared. Night-vision goggles and IR temperature sensing use far infrared wavelengths of 2000-10,000 nm, depending on the application. Table 7-1 shows the wavelengths for common colors along with the infrared spectrum.

Table 7-1: Colors and Approximate Wavelengths

Color	Wavelength	Color	Wavelength
Violet	400	Red	780
Blue	470	Near infrared	800-1000
Green	565	Infrared	1000-2000
Yellow	590	Far infrared	2000-10,000
Orange	630		

Infrared Headlights

The infrared object detection system we'll build on the Boe-Bot is like a car's headlights in several respects. When the light from a car's headlights reflects off obstacles, your eyes detect the obstacles and your brain processes them and makes your body guide the car accordingly. The Boe-Bot uses infrared LEDs for headlights as shown in Figure 7-1. They emit infrared, and in some cases, the infrared reflects off objects and bounces back in the direction of the Boe-Bot. The eyes of the Boe-Bot are the infrared detectors. The infrared detectors send signals indicating whether or not they detect infrared reflected off an object. The brain of the Boe-Bot, the BASIC Stamp, makes decisions and operates the servo motors based on this sensor input.

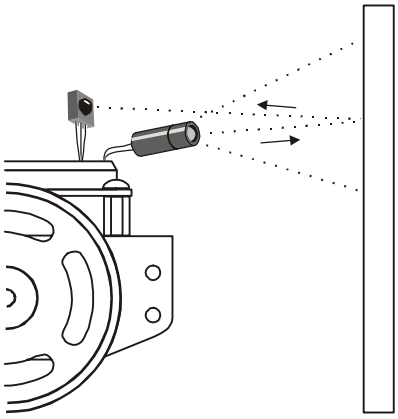


Figure 7-1
Object Detection
with IR Headlights

The IR detectors have built-in optical filters that allow very little light except the 980 nm infrared that we want to detect with its internal photodiode sensor. The infrared detector also has an electronic filter that only allows signals around 38.5 kHz to pass through. In other words, the detector is only looking for infrared that's flashing on and off 38,500 times per second. This prevents IR interference from common sources such as sunlight and indoor lighting. Sunlight is DC interference (0 Hz), and indoor lighting tends to flash on and off at either 100 or 120 Hz, depending on the main power source in the region. Since 120 Hz is outside the electronic filter's 38.5 kHz band pass frequency, it is completely ignored by the IR detectors.

Some fluorescent lights do generate signals that can be detected by the IR detectors. 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.

ACTIVITY #1: BUILDING AND TESTING THE IR PAIRS

In this activity, you will build and test the infrared transmitter/detector pairs.

Parts List:

(2) Infrared detectors
(2) IR LEDs (clear case)
(2) IR LED shield assemblies
(2) Resistors - 220 Ω (red-red-brown)
(2) Resistors – 1 kΩ (brown-black-red)

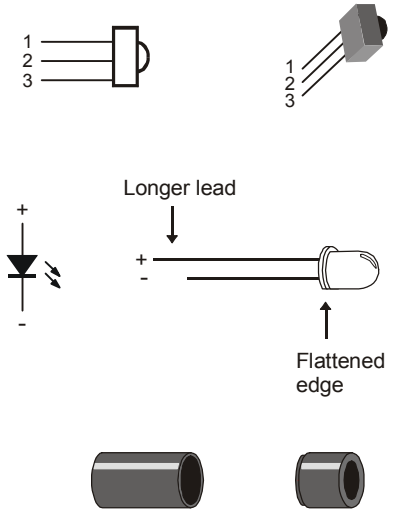


Figure 7-2 New Parts Used in this Chapter

IR detector (top)

IR LED (middle)

IR LED shield assembly (bottom)

Building the IR Headlights

√ Insert the infrared LED into the shield assembly as shown in Figure 7-3.
√ Make sure the LED snaps into the larger part of the housing.
√ Snap the smaller part of the housing over the LED case and onto the larger part.

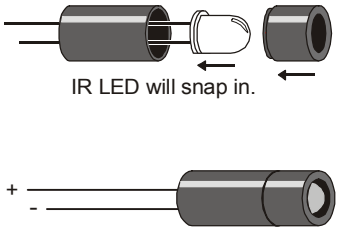


Figure 7-3 Snapping the IR LED into the Shield Assembly

One IR pair (IR LED and detector) is mounted on each corner of the breadboard. Figure 7-4 shows the IR headlights circuit as a schematic and Figure 7-5 shows the circuit as a wiring diagram.

- √ Disconnect power from your board and servos.
- √ Build the circuit shown by the schematic in Figure 7-4, using the wiring diagram for your board in Figure 7-5 as a reference for parts placement.

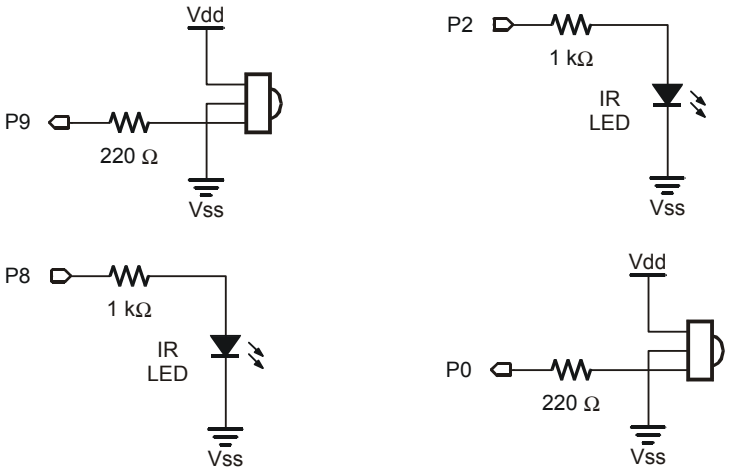


Figure 7-4
Left and Right IR Pairs

Watch your IR LED anodes and cathodes!

Remember that the anode lead is the longer lead on an IR LED by convention, but that you need to check the LED's plastic case to make sure. The cathode lead is the one near the flat spot on the case. In Figure 7-5, 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 connects to Vss with a jumper wire.

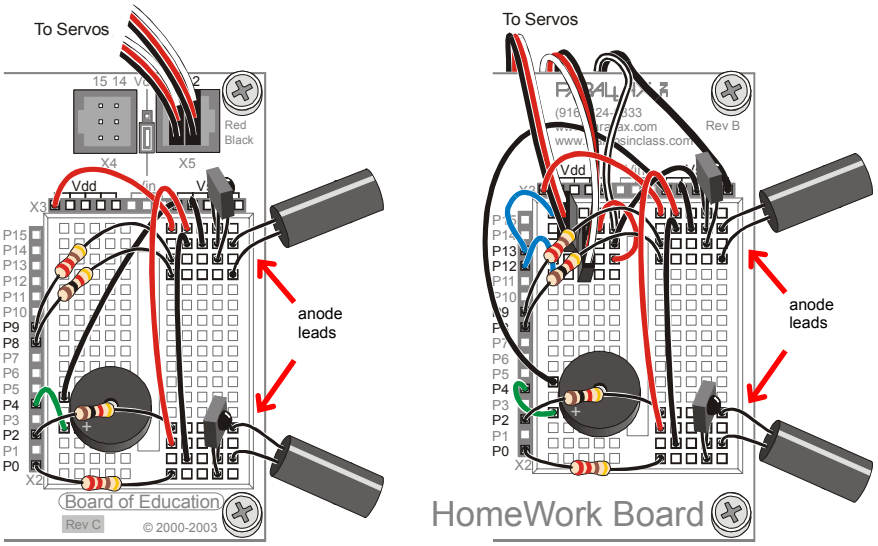


Figure 7-5 Wiring Diagrams for Infrared Emitter and Receiver Circuits

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

7

Testing the IR Pairs Using the FREQOUT Trick

The **`FREQOUT`** command was designed mainly to synthesize audio tones. The actual range of the **`FREQOUT`** command is 1 to 32768 Hz. One interesting phenomenon of digitally synthesized tones is that they contain signals called harmonics. A harmonic is a higher frequency tone that's mixed in with the tone you want to hear. These tones are outside human abilities to detect sound, which tend to range from 20 Hz to 20 kHz. The harmonics generated by the **`FREQOUT`** command start at 32769 Hz and go upward. You can directly control these harmonics using ***`Freq1`*** arguments above 32768. In this activity, you will use the command **`FREQOUT 8, 1, 38500`** to send a 38.5 kHz harmonic that lasts 1 ms to P8. The infrared LED circuit connected to P8 will broadcast this harmonic. If the infrared light is reflected back to the Boe-Bot by an object in its path, the infrared detector will send the BASIC Stamp a signal to let it know that the reflected infrared light was detected.

FREQOUT Command - Fundamentals and Harmonics

The fundamental frequency is the value of the ***Freq1*** argument when it's at or below 32768. For example, when you use the command **FREQOUT 4, 2000, 3000**, the fundamental frequency is 3000 Hz. That's the intended sound, but there is also a harmonic sound that accompanies it. This harmonic is a much higher frequency that the human ear can detect, in the neighborhood of 62.5 kHz. Here's how to calculate the harmonic frequency given the fundamental and visa versa.

Whenever you use the **FREQOUT** command to send a tone in this range, it contains that hidden (harmonic) tone as well. The equation for the harmonic is:

harmonic frequency = 65536 – Freq1, Freq1 <= 32678

Whenever you use the **FREQOUT** command with a ***Freq1*** argument above 32768 to send a harmonic, it contains a fundamental tone. The equation for the fundamental is:

fundamental frequency = 65536 – Freq1, Freq1 > 32768

The key to making each IR LED/detector pair work is to send 1 ms of 38.5 kHz **FREQOUT** harmonic, and then, immediately store the IR detector's output in a variable. Here is an example that sends the 38.5 kHz signal to the IR LED connected to P8, then stores the IR detector's output, which is connected to P9, in a bit variable named **irDetectLeft**.

```
FREQOUT 8, 1, 38500
irDetectLeft = IN9
```

The IR detector's output state when it sees no IR signal is high. When the IR detector sees the 38500 Hz harmonic reflected by an object, its output is low. The IR detector's output only stays low for a fraction of a millisecond after the **FREQOUT** command is done sending the harmonic, so it's essential to store the IR detector's output in a variable immediately after sending the **FREQOUT** command. The value stored by the variable can then be displayed in the Debug Terminal or used for navigation decisions by the Boe-Bot.

Example Program: TestLeftIrPair.bs2

- √ Reconnect power to your board.
- √ Enter, save, and run TestLeftIrPair.bs2.

```
' Robotics with the Boe-Bot - TestLeftIrPair.bs2
' Test IR object detection circuits, IR LED connected to P8 and detector
' connected 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 the serial cable, because you will be using the Debug Terminal to test your IR pair.
- √ Place an object, such as your hand or a sheet of paper, about an inch from the left IR pair, in the manner shown in Figure 7-1 on page 236.
- √ Verify that when you place an object in front of the IR pair the Debug Terminal displays a 0, and when you remove the object from in front of the IR pair, it displays a 1.
- √ If the Debug Terminal displays the expected values for object not detected (1) and object detected (0), move on to the Your Turn section following the example program.
- √ If the Debug Terminal does not display the expected values, try the steps in the Trouble-Shooting box.

Trouble-Shooting

If the Debug Terminal does not display the expected values, check for circuit and program entry errors.

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.

Your Turn

- √ Save TestLeftIrPair.bs2 as TestRightIrPair.bs2.

- √ Change the **DEBUG** statement, title and comments to refer to the right IR pair.
- √ 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 right IR pair; with the IR LED circuit connected to P2 and the detector connected to P0.

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 7-6 using Figure 7-7 as a reference.

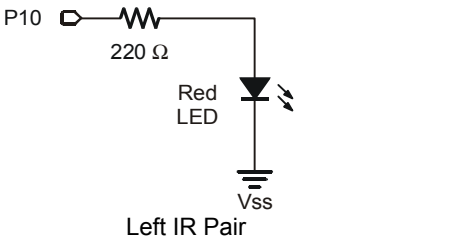


Figure 7-6
Left and Right Indicator LEDs

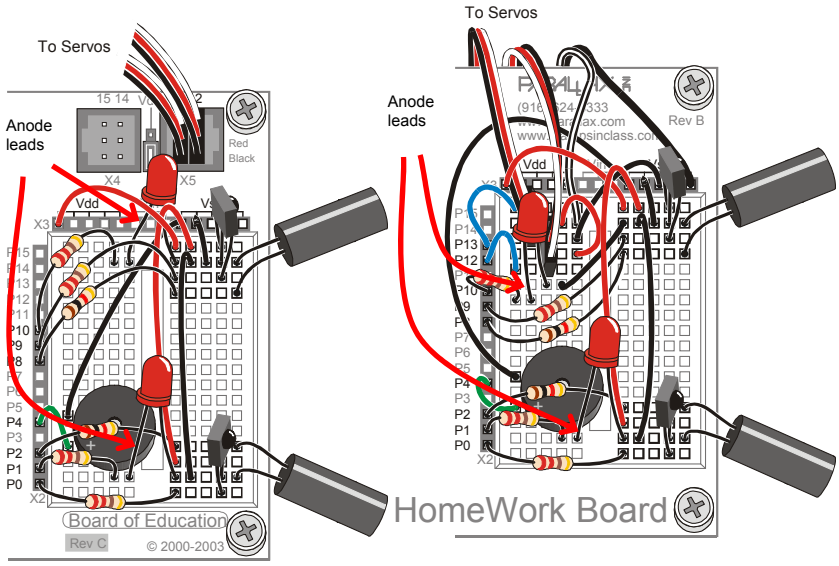


Figure 7-7
Wiring Diagrams for Red LED Indicators with IR Object Detection Circuits

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

7

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 serial cable and testing other objects.

Example Program – TestIrPairsAndIndicators.bs2

- √ Reconnect power to your board.
- √ Enter, save, and run TestIrPairsAndIndicators.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 - TestIrPairsAndIndicators.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 serial cable.

- √ Unplug your Boe-Bot from the serial 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 before-hand.

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 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. You can do this with a separate Boe-Bot that's running TestIrPairsAndIndicators.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 serial 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 LED in the circuit connected to P1 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 TestIrPairsAndIndicators.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 facing the IR LED can be detected, with the 1 kΩ resistors in place, and record your data in Table 7-2.
- √ 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.
√ Repeat with 470 Ω resistors.
√ Repeat with 220 Ω resistors.

Table 7-2: Detection Distance vs. Resistance

IR LED Series Resistance, (Ω)	Maximum Detection Distance, Circle one: (in / cm)
4700	
2000	
1000	
470	
220	

√ Before moving on to the next activity, restore your IR pairs 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 TestIrPairsAndIndicators.bs2 to verify that both IR LED/detector pairs 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 pairs.

```
FREQOUT 8, 1, 38500
```

```
irDetectLeft = IN9
```

The **IF…THEN** statements were modified so that they look at the variables that store the IR pair 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 pairs.

' {$STAMP BS2}                              ' Stamp directive.
' {$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 IR pairs detect obstacle
    GOSUB Turn_Left                            ' Back up & U-turn (left twice)
    GOSUB Turn_Left
  ELSEIF (irDetectLeft = 0) THEN               ' Left IR pair detects
    GOSUB Back_Up                              ' Back up & turn right
    GOSUB Turn_Right
  ELSEIF (irDetectRight = 0) THEN              ' Right IR pair detects
    GOSUB Back_Up                              ' Back up & turn left
    GOSUB Turn_Left
  ELSE                                         ' Both IR pairs 1, no detects
    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 pairs 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 7-8. The program should make it continue forward so long as both IR detectors can "see" the surface of the table. In other words, the Boe-Bot can continue forward so long as the table top it's navigating on is detected.

- √ Disconnect power from your board and servos.
- √ Point your IR pairs downward and outward as shown in Figure 7-8.

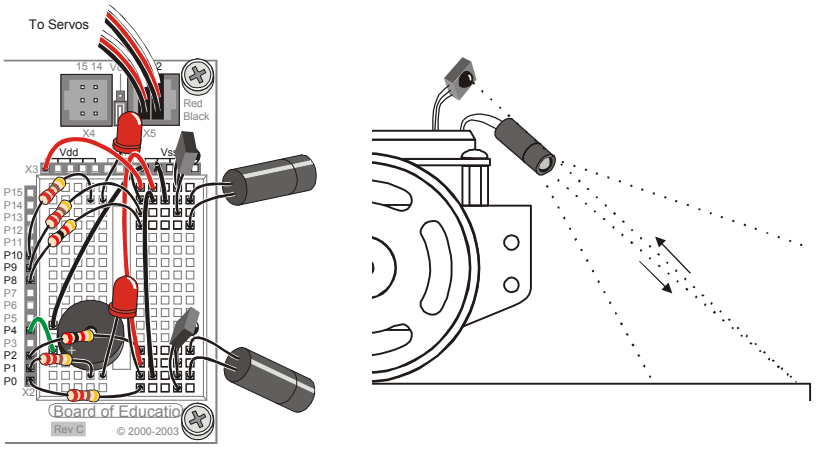


Figure 7-8
IR Pairs Directed Downwards to Scan for a Drop-Off

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 7-9. 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 246) to make sure that nearby fluorescent lighting will not interfere with your Boe-Bot's IR detectors.
- √ Use the TestIrPairsAndIndicators.bs2 (page 244) 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.

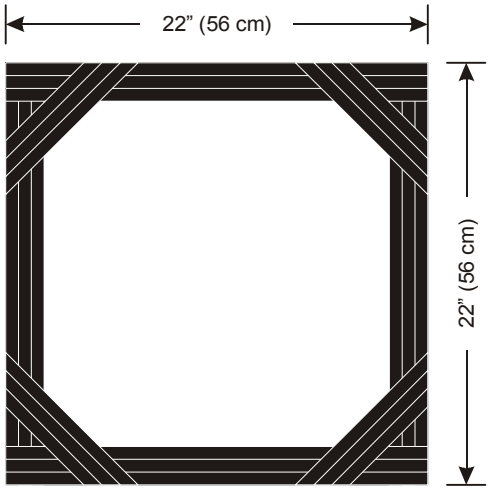


Figure 7-9
Electrical Tape Outline Simulates Tabletop Edge

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 the values of **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** variable.

```
  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 harmonic sent by **`FREQOUT 2, 1, 38500`**? What is the value of the fundamental 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 LED/detector pairs.
2. Explain the function of `pulseCount` in AvoidTableEdge.bs2. How does this relate to your answer to Exercise 3?

Projects

1. Design a Boe-Bot application that sits still until you wave your and 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, then continues roaming. This alarm should be different from the low battery alarm.

Solutions

Q1. 38.5 kHz is the frequency of the harmonic. Its fundamental 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
```

This will only monitor the right detector.

E2. The **`pulseCount`** variable allows the Boe-Bot to have adjustable distance of motion depending on the situation.

P1. The FastIrRoaming.bs2 program can be combined with a **`DO…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 sensors 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 237.

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). 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.

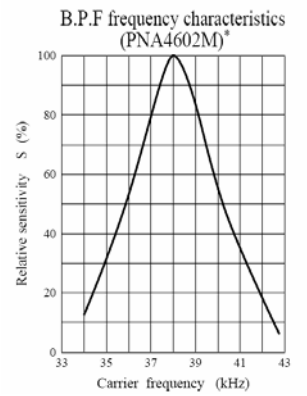


Figure 8-1
Filter Sensitivity Depends on Carrier Frequency

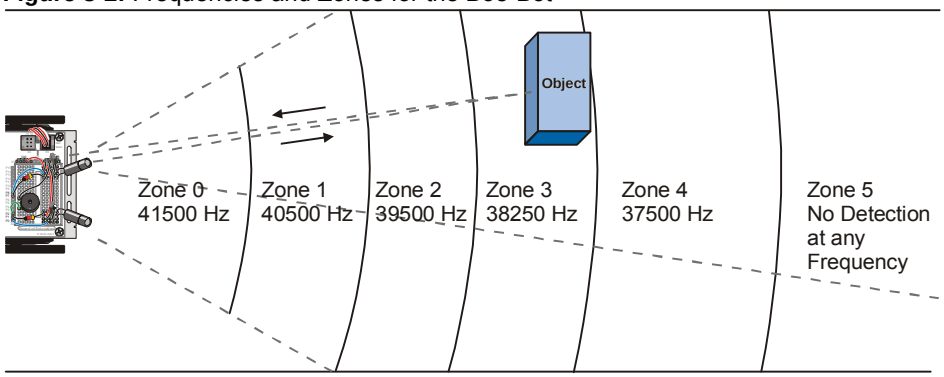
Another way to think about it is that the most sensitive frequency will detect the objects that are the farthest away, while less sensitive frequencies can only be used to detect closer objects. This makes distance detection simple. Pick 5 frequencies, then test them from most sensitive to least sensitive. 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: 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 (harmonic) 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. These frequencies were determined using tests similar to the ones outlined in Appendix G: Tuning IR Distance Detection.

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** operator. 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 LED/detector pair (connected to P8 and P9) to make sure they are 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.

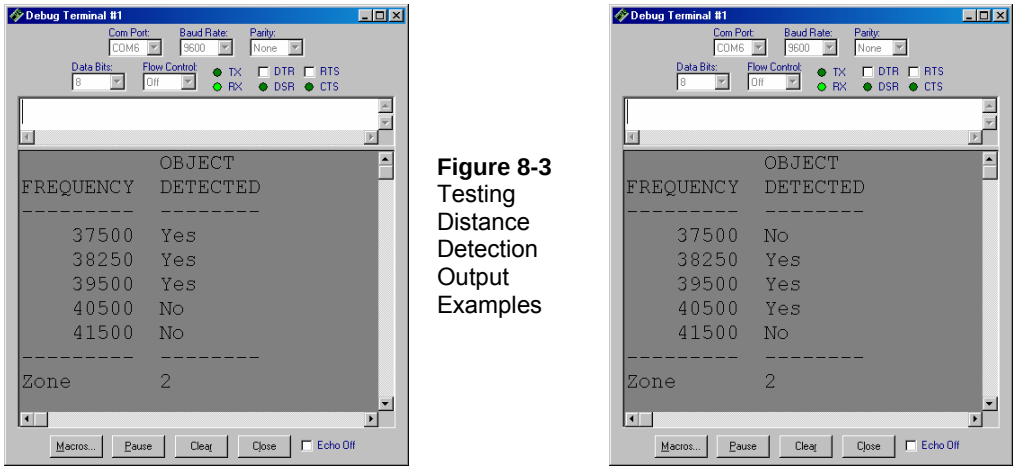


Figure 8-3 Testing Distance Detection Output Examples

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.

- √ 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.

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. Older shrink wrap LED distances will be less. 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, which is likely if you have shrink wrapped IR LEDS, 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 Pair

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 LED/detector pair.
- √ Run the program and verify that this pair 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 LED/detector pairs 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.

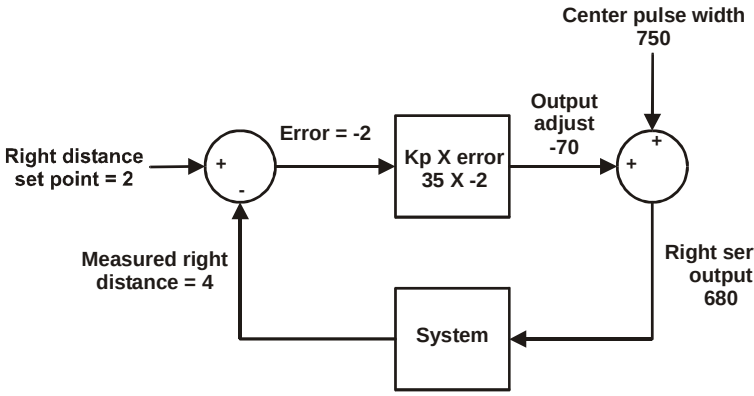


Figure 8-4
Proportional Control Block Diagram for Right Servo and IR LED and Detector Pair

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.

Error = *Right distance set point – Measured right distance*

	=	*2 – 4*
Output adjust	=	*error* $\times K_p$
	=	*–2 × 35*
	=	*– 70*
Right servo output	=	*Output adjust + Center pulse width*
	=	*– 70 + 750*
	=	*680*

By making some substitutions, the three equations above can be reduced to this one, which will give you the same result.

Right servo output	=	*(Right distance set point – Measured right distance) × Kp + Center pulse width*

By substituting the values from the example, we can see that the equation still works:

	=	*((2 – 4) × 35) + 750*
	=	*680*

The left servo and IR pair 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 pair, 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.

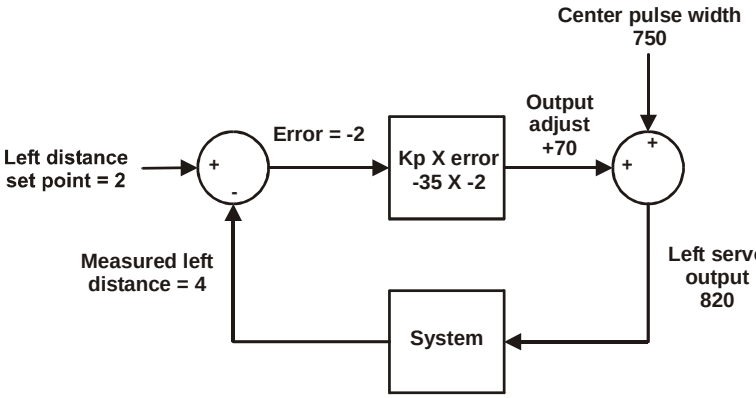


Figure 8-5
Proportional Control Block Diagram for Left Servo and IR LED and Detector Pair

Programming the Boe-Bot Shadow Vehicle

Remember that the equation for the right servo's output was:

$$\textit{Right servo output} = \textit{(Right distance set point} - \textit{Measured right distance)} \times \textit{Kp} + \textit{Center pulse width}$$

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`**, and after that, the product is added to the center pulse width.

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 × 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 the FollowingBoeBot.bs2 Works

FollowingBoeBot.bs2 declares four constants, **Kpr**, **Kpl, SetPoint**, and **CenterPulse** using the **CON** directive. Everywhere you see **SetPoint**, it's actually the number 2 (a constant). Likewise, everywhere you see either **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 pairs.

```
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.

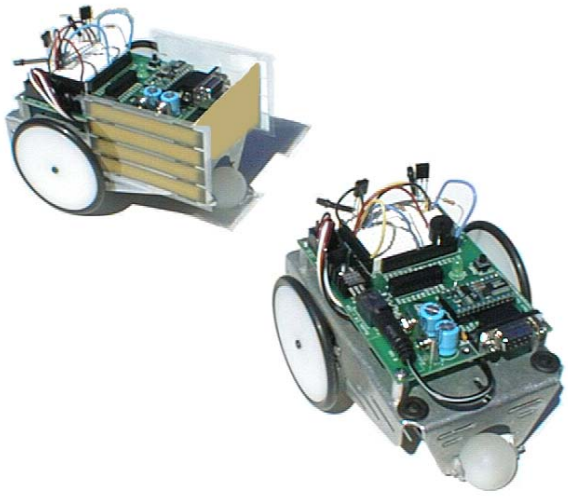


Figure 8-6
Lead Boe-Bot (left) and Shadow Boe-Bot (right)

- √ 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. Open FastIrRoaming.bs2, and rename it SlowerIrRoamingForLeadBoeBot.bs2.
- √ Make these modifications to SlowerIrRoamingForLeadBoeBot.bs2:
  - √ Increase all **`PULSOUT Duration`** arguments that are now 650 to 710.
  - √ 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.

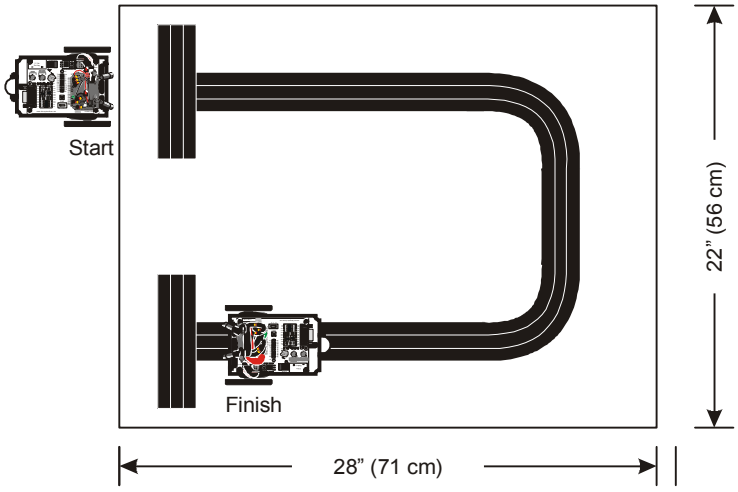


Figure 8-7
Stripe Following Course

Building and Testing the Course

For successful navigation of this course, some testing and Boe-Bot adjustment will be required.

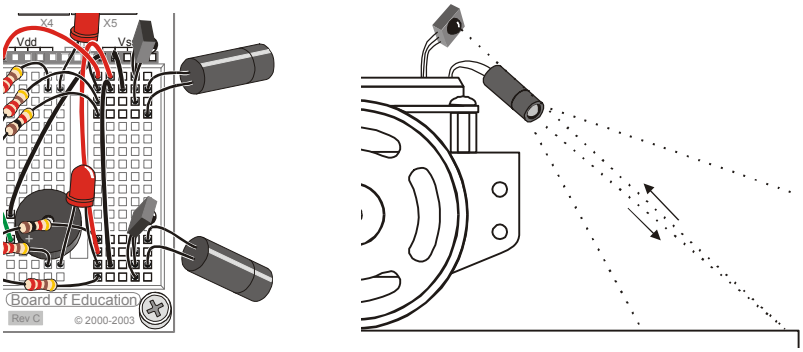
Materials Required

(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.

√ Use your poster board and electrical tape to build the course shown in Figure 8-7.

Testing the Stripe

√ Point your IR pairs downward and outward as shown in Figure 8-8 (Figure 7-8 from page 255 repeated here for convenience).



Top View Side View

Figure 8-8
IR Pairs Directed Downwards to Scan for the Stripe

√ Make sure your electrical tape course is free of fluorescent light interference. See Sniffing for IR Interference (page 245).
√ 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 serial 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.

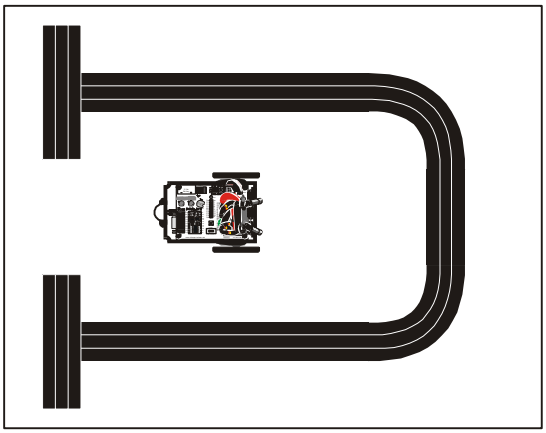


Figure 8-9
Test for Low Zone Number – Top View

√ Place your Boe-Bot so that both IR LED/detector pairs 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.

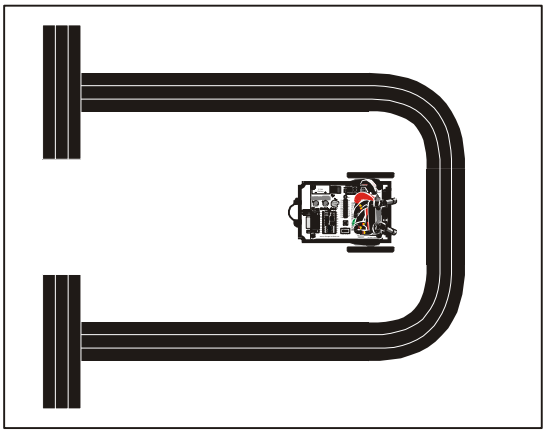


Figure 8-10
Test for High Zone Number – Top View

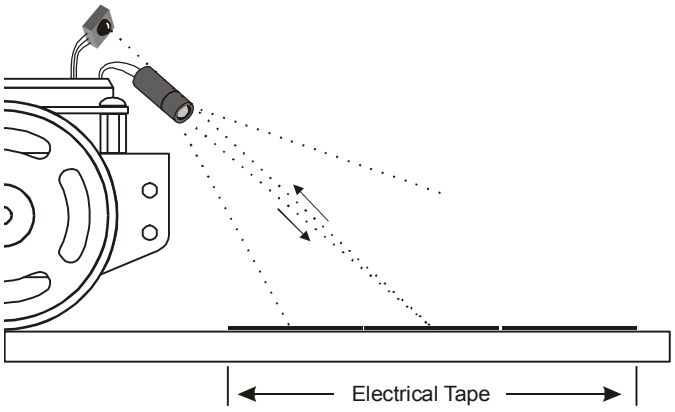


Figure 8-11
Test for High Zone Number – Side View

Trouble Shooting 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).

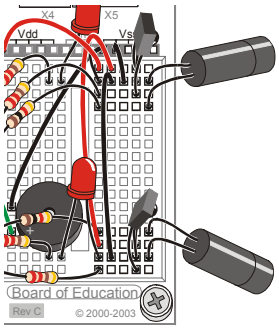
If you are using the older shrink wrapped IR LEDs instead of the ones with the 2-piece plastic shields, you may be having trouble with getting a low zone number when the IR LED/detectors are focused on the white background. These LEDs may need 470 Ω (yellow-violet-brown) or 220 Ω (red-red-brown) resistors in series. Also, make sure that the leads of the IR LEDs are not touching each other.

√ 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 pairs 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.

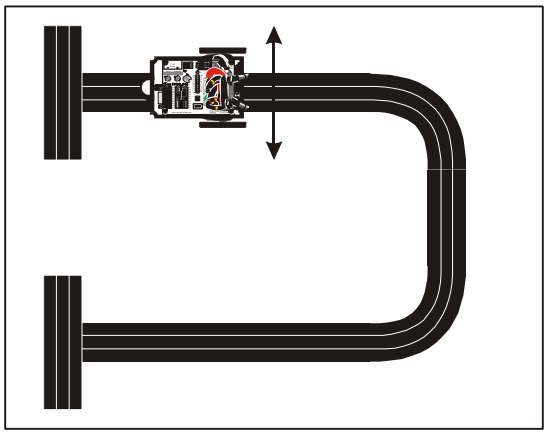
When you move the Boe-Bot in either direction indicated by the double-arrow, one or the other IR pair will become focused on the electrical tape. When you do this, the readings

for the pair 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 LED/detector pairs until the Boe-Bot passes this last test. Then you will be ready to try following the stripe.



IR pairs close-up



Top view of Boe-Bot straddling the stripe

Figure 8-12
Stripe Scan Test

Programming for Stripe Following

You will only need to make a few small adjustments to FollowingBoeBot.bs2. from page 282 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 (shown below).
- √ 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 pairs. 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.

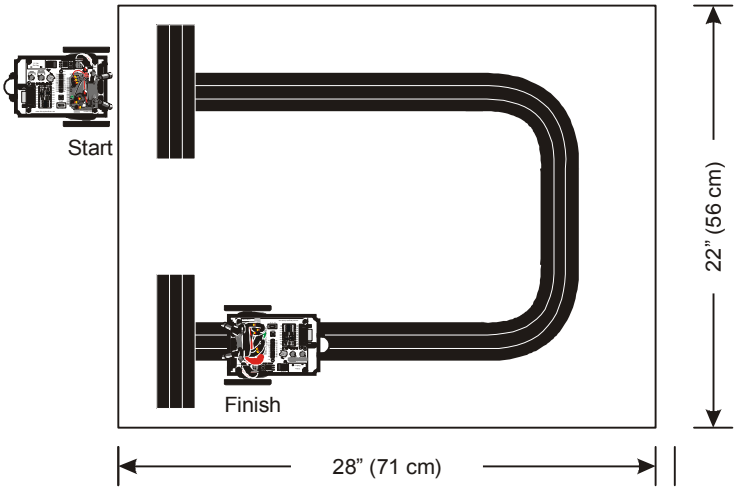


Figure 8-13
Stripe Following Course


```
' -----[ 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.

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.

Watch the Boe-Bot in Action at www.parallax.com!

You can see the Boe-Bot solving Chapter 8 Project 2 and other Robotics video clips in the Robo Video Gallery under the Robotics Menu at www.parallax.com.

Questions

1. What would the relative sensitivity of the IR detector be if you use **FREQOUT** to send a 35 kHz harmonic? What is the relative sensitivity with a 36 kHz harmonic?
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

P1. 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.

P2. *Advanced Project* - Design a maze-solving contest of your own, and program the Boe-Bot to solve it!

Questions

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
```

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 program 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                           ' Initialze Boolean variable
  numPulses = numPulses + 1             ' Count total pulses sent

  SELECT numPulses
    CASE < 60
      IF (pulseLeft > CenterPulse) THEN
        fwdPulses = fwdPulses + 1       ' Count foward 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 pairs 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 StampsInClass Yahoo! Group, listed behind the title page of Robotics with the Boe-Bot. Or, you can email the parallax Educational Team directly at stampsinclass@parallax.com.

A

Appendix A: PC to BASIC Stamp Communication Trouble-Shooting

Here is a list of things to try to quickly fix any difficulties getting the BASIC Stamp Editor to communicate with your BASIC Stamp:

- √ If you are using a Board of Education Rev C, make sure the power switch is set to position-1.
- √ Rule out dead batteries and incorrect or malfunctioning power supplies by using a new 9 V battery or four new 1.5 V AA alkaline batteries in the battery pack.
- √ Make sure the serial cable is firmly connected to both the computer's COM port and the DB9 connector on the Board of Education or BASIC Stamp HomeWork Board.
- √ Make sure that your serial cable is a normal (straight-through) serial cable. DO NOT USE A NULL MODEM CABLE. Most null modem cables are labeled NULL or Null Modem; visually inspect the cable for any such labeling.
- √ Disable any palmtop communication software.

If you are using a BASIC Stamp and Board of Education, also check the following:

- √ Make sure the BASIC Stamp was inserted into the socket right-side-up as shown in Figure 1-24 on page 17.
- √ If you are using a DC power supply that plugs into the wall, make sure it is plugged in to both the wall and the Board of Education. Verify that the green Pwr light on the Board of Education emits light when the DC supply is plugged in.
- √ Make sure the BASIC Stamp is firmly inserted into the socket.
- √ Visually inspect the BASIC Stamp module to make sure that none of the pins folded under the module instead of sinking into their sockets on the Board of Education.

If your Identification window looks similar to Figure A-1, it means that the BASIC Stamp Editor cannot find your BASIC Stamp on any COM port. If you have this problem, try the following:

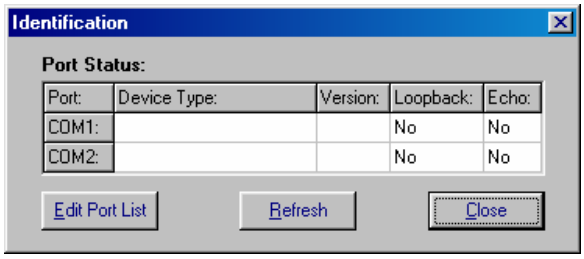


Figure A-1
Identification Window

Example: BASIC Stamp 2 not found on COM ports.

If you know the number of the COM port, but it does not appear in the Identification Window:

- √ Use the Edit Port List button to add that COM port. When you return to the Identification window, click the Refresh button to find out if the BASIC Stamp 2 is now detected.
- √ Close the Identification window.
- √ In the BASIC Stamp Editor, Click Edit and select Preferences. Click the Editor Operation tab, and set the Default COM Port to AUTO.
- √ Try the Run → Identify test again.

If you are unsure of which COM port your BASIC Stamp is connected to, or if you are using a USB to serial port adaptor, you may need to look in your computer's Device Manager to find the list of COM ports in use.

- √ Click on your computer desktop's Start button.
- √ To view the list of COM ports in use, make the selections listed next to your operating system :

Windows® 98:	Control Panel → System → Hardware → Device Manager → Ports(COM & LPT1).
Windows® 2000:	Settings → Control Panel → System → Hardware → Device Manager → Ports (COM & LPT).
Windows® XP:	Control Panel → Printers and Other Hardware. In the See Also box select System. Select Hardware → Device Manager → Ports
Windows® XP Pro:	Settings → Control Panel → System → Hardware → Device Manager → Ports (COM & LPT).

- √ If you are using a serial port (no USB to serial adaptor), make a note of the COM ports listed. If one or more of these COM ports do not appear in your BASIC Stamp Editor's list, make a note of the numbers for each COM port that doesn't appear in the list now.
- √ If you are using an FTDI USB to Serial adaptor, look for the COM port that reads FTDI USB to Serial COM…
- √ Repeat the Run → Identify test.
- √ Click the Edit Ports List button and add the missing COM port numbers.
- √ Repeat the Run → Identify test again, this time, the Identification window should "find" your BASIC Stamp 2.

Still no BASIC Stamp Detected?

- √ If you have more than one COM port, try connecting your Board of Education or BASIC Stamp HomeWork Board to a different COM port and see if Run → Identify works then.
- √ If you have a second computer, try it on the different computer.

If you get the error message "No BASIC Stamp Found" but the Run → Identify test shows a "Yes" in both columns for one of the COM ports, you may need to change a setting to your FIFO Buffers. This happens occasionally with Microsoft Windows® 98 and XP users. Make a note of the COM port with the "Yes" messages, and try this:

Windows® 98:

- √ Click on your computer desktop's Start button.
- √ Select Settings→ Control Panel → System → Device Manager → Ports (COM & LPT).
- √ Select the COM port that was noted by the Run → Identify test.
- √ Select Properties → Port Settings → Advanced.
- √ Uncheck the box labeled "Use FIFO Buffers" then click OK.
- √ Click OK as needed to close each window and return to the BASIC Stamp Editor.
- √ Try downloading a program once more.

Windows® 2000:

- √ Click on your computer desktop's Start button.
- √ Select Settings → Control Panel → System → Hardware → Device Manager → Ports (COM & LPT).

- √ Select the COM port that was noted by the Run → Identify test.
- √ Select → Port Settings → Advanced.
- √ Uncheck the box labeled "Use FIFO Buffers" then click OK.
- √ Click OK as needed to close each window and return to the BASIC Stamp Editor.
- √ Try downloading a program once more.

Windows® XP:

- √ Click on your computer desktop's *Start* button.
- √ Select Control Panel → Printers and Other Hardware.
- √ In the See Also box select System.
- √ Select Hardware → Device Manager → Ports.
- √ Enter the COM port number noted by the Run→ Identify test.
- √ Select Port Settings → Advanced.
- √ Uncheck the box labeled "Use FIFO Buffers" then click OK.
- √ Click OK to close each window as needed and return to the BASIC Stamp Editor.
- √ Try downloading a program once more.

Windows® XP Pro:

- √ Click on your computer desktop's *Start* button.
- √ Select Control Panel → System → Hardware → Device Manager → Ports(COM & LPT1).
- √ Select the Communications Port number noted by the Run → Identify test.
- √ Select Properties → Port Settings → Advanced.
- √ Uncheck the box labeled "Use FIFO Buffers" then click OK.
- √ Click OK to close each window as needed and return to the BASIC Stamp Editor.
- √ Try downloading a program once more.
- √

If none of these solutions work, you may go to www.parallax.com and follow the Support link. Or, email support@parallax.com or call Tech Support toll free at 1-888-99-STAMP.

Appendix B: BASIC Stamp and Carrier Board Components and Features

B

The BASIC STAMP® 2 Microcontroller Module

Figure B-1 shows a close-up of the BASIC Stamp® 2 microcontroller module. Its major components and their functions are indicated by labels.

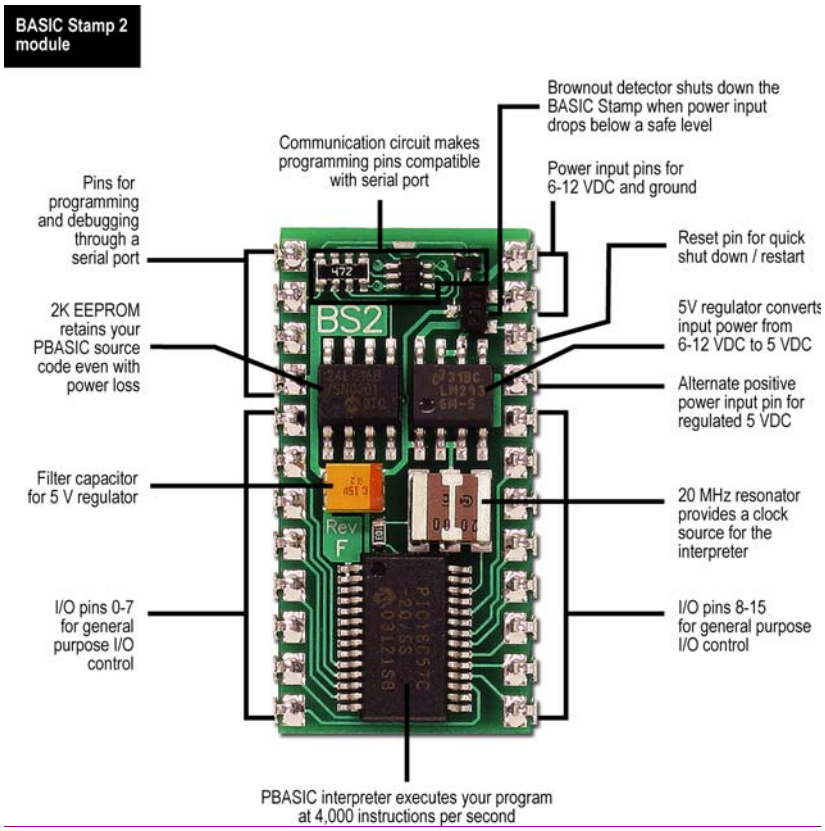


Figure B-1: BASIC Stamp® 2 Microcontroller Module Components and Their Functions

The Board of Education® Rev C Carrier Board

The Board of Education® Rev C carrier board for BASIC Stamp® 24-pin microcontroller modules is shown in Figure B-2. Its major components and their functions are indicated by labels.

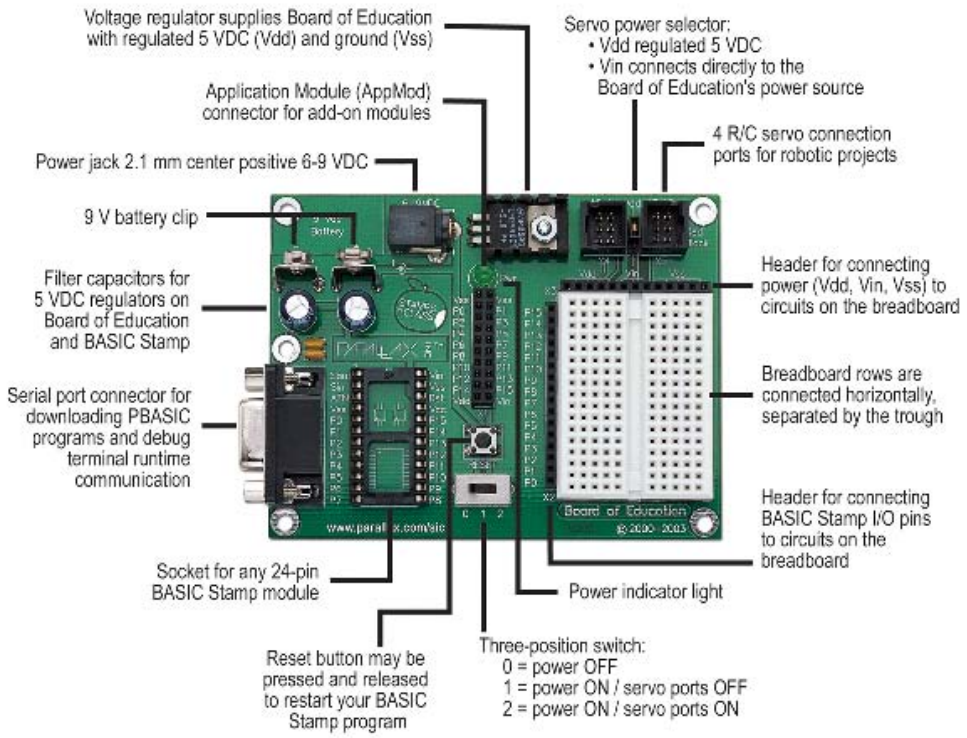


Figure B-2: Board of Education® Rev C Carrier Board

B

The BASIC Stamp® HomeWork Board™ Project Platform

The BASIC Stamp® HomeWork Board™ project platform is shown in Figure B-3. Its major components and their functions are indicated by labels.

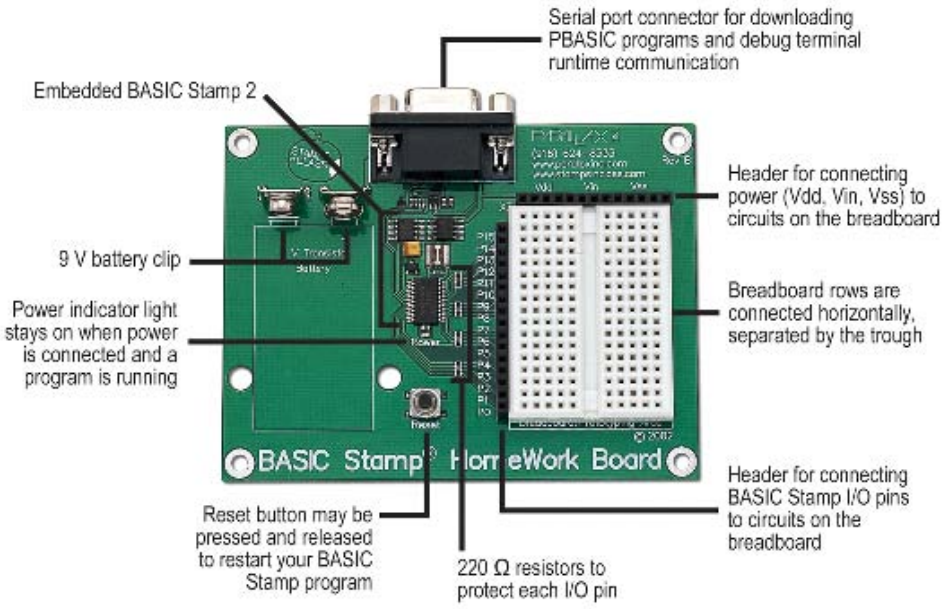


Figure B-3: BASIC Stamp® HomeWork Board™ Project Platform

The Board of Education® Rev B Carrier Board

Figure B-4 shows the Board of Education® Rev B carrier board for BASIC Stamp® 24-pin microcontroller modules. Its major components and their functions are indicated by labels.

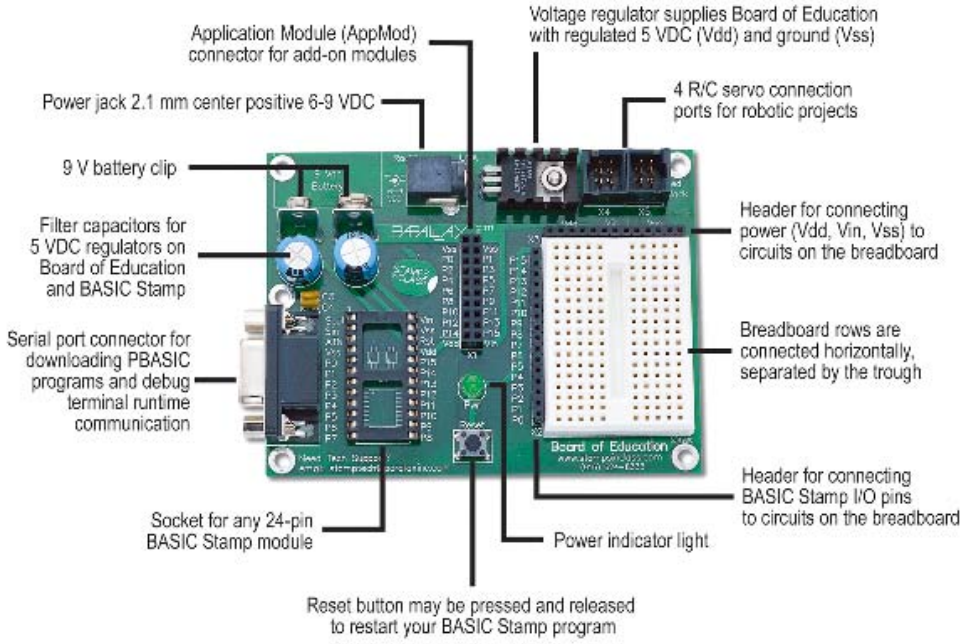


Figure B-4: Board of Education® Rev B Carrier Board

Appendix C: 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. For example, the color code for the 470 Ω resistor is yellow-violet-brown.

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

Each color bar that tells you the resistor's value corresponds to a digit, and these colors/digits are listed in Table C-1. Figure C-1 shows how to use each color bar with the table to determine the value of a resistor.

Table C-1: Resistor Color Code Values

Digit	Color
0	Black
1	Brown
2	Red
3	Orange
4	Yellow
5	Green
6	Blue
7	Violet
8	Gray
9	White

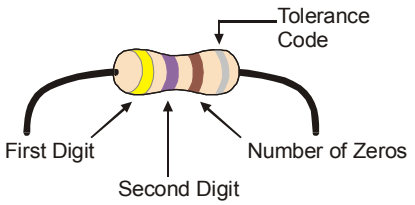


Figure C-1 Resistor Color Codes

Here is an example that shows how Table C-1 and Figure C-1 can be used to figure out a resistor value by proving that yellow-violet-brown is really 470 Ω:

- 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.

Yellow-Violet-Brown = 4-7-0.

Appendix D: 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 D-1).

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.

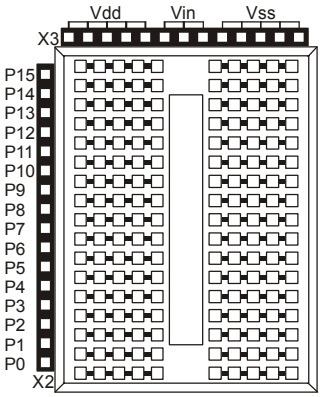


Figure D-1
Prototyping Area

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 D-2. For each of these parts, the part drawing is shown above the schematic symbol.

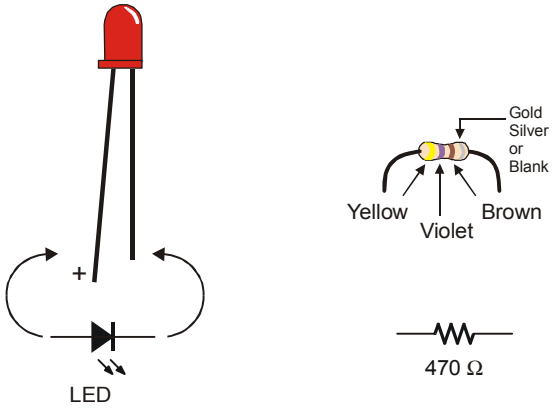


Figure D-2
Part Drawings and Schematic Symbols

LED(left) and 470 Ω resistor (right)

Figure D-3 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.

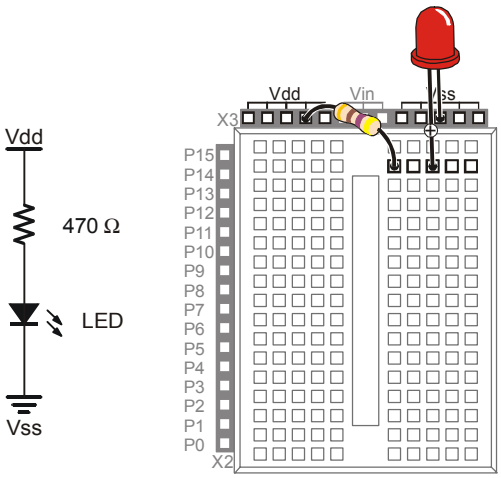


Figure D-3
Example Schematic and Wiring Diagram

Schematic (left) and wiring diagram (right)

D

Figure D-4 shows a second example of a schematic and wiring diagram. This schematic shows P14 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. The schematic only differs by 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, mainly 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. The wiring diagram shows how this difference is handled with the resistor lead that used to be plugged into Vdd, now plugged into P14.

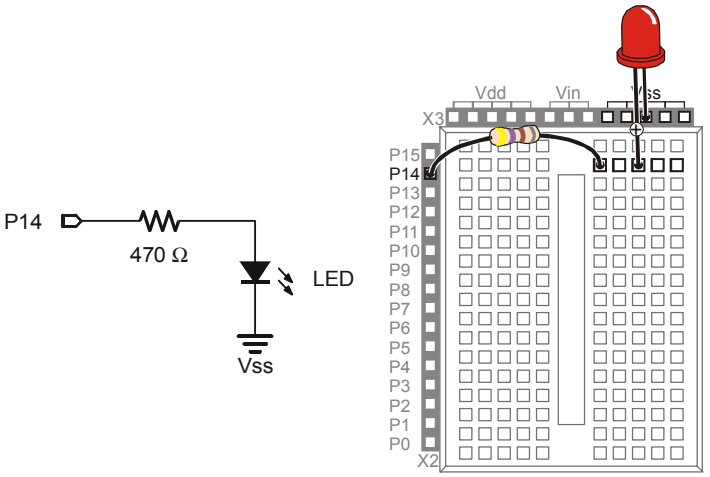


Figure D-4
Example Schematic and Wiring Diagram

Schematic (left) and wiring diagram (right)

Here is a more complex example that involves two additional parts, a photoresistor and a capacitor. The schematic symbols and part drawings for these components are shown in Figure D-5.

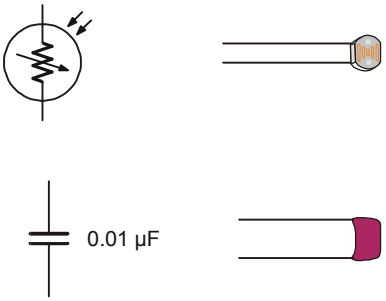


Figure D-5
Part Drawings and Schematic Symbols

Photoresistor (top) and non-polar capacitor (bottom)

Since this schematic shown in Figure D-6 calls for a 220 Ω resistor, the first step is to consult Appendix C: Resistor Color Codes to determine the color code for a 220 Ω resistor. The color code is Red, Red, Brown. 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 D-7). In the schematic, the other lead of the resistor is connected to not one, but two other component terminals. A terminal from the photoresistor and capacitor both share this connection. On the breadboard, the other resistor lead is plugged into one of the rows of 5 sockets. This row also has leads from the capacitor and photoresistor plugged into it. In the schematic, the other terminals of the photoresistor and capacitor 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 leads for the capacitor and photoresistor were plugged into the same row, completing the circuit.

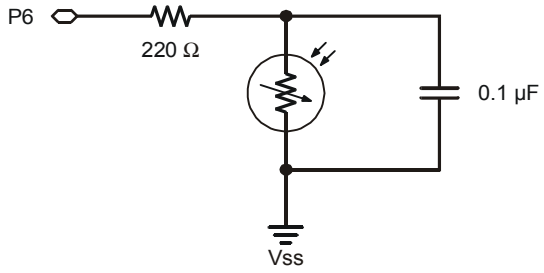


Figure D-6
Resistor, Photoresistor, and Capacitor Schematic

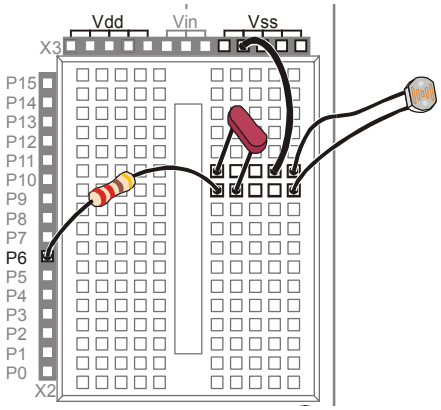


Figure D-7
Resistor, Photoresistor, and Capacitor Wiring Diagram

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 D-8 shows another solution to the schematic just discussed. Follow the connections and convince yourself that it does satisfy the schematic.

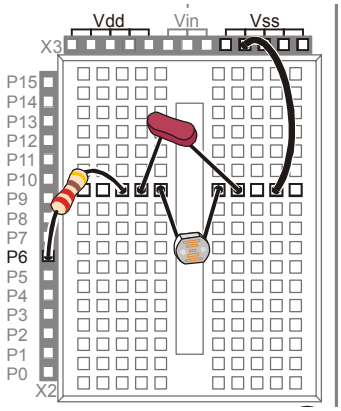


Figure D-8
Resistor, Photoresistor, and Capacitor Wiring Diagram

Note the alternative parts placement.

Appendix E: Boe-Bot Parts Lists

To complete the activities in this text, you will need a complete Boe-Bot and the electronic components necessary to build the example circuits. There are several options for ordering these items from Parallax, which are described on the following pages.

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 and free downloads about your Boe-Bot and the Robotics with the Boe-Bot Student Guide, check their individual product pages at www.parallax.com.

Boe-Bot Robot Kit (also known as the Boe-Bot Full Kit)

Aside from a PC with a serial port and a few common household items, the Boe-Bot Robot Kit contains all the parts and documentation you'll need to complete the experiments in this text.

Table E-1: Boe-Bot Robot (Full) Kit (#28132)
Parts and quantities subject to change without notice

Parallax Stock Code	Description	Quantity
BS2-IC	BASIC Stamp 2 microcontroller module	1
27000	Parallax CD with software and documentation	1
28124	Robotics with the Boe-Bot Parts Kit	1
28125	Robotics with the Boe-Bot Student Guide	1
28150	Board of Education Rev C	1
700-00064	Parallax Screwdriver	1
800-00003	Serial cable	1

All of these items may also be ordered separately, using the individual part numbers. You can contact the Parallax Sales Team toll free at 1-888-512-1024 or order online at www.parallax.com. For technical questions or assistance call our Technical Support team at 1-888-99-STAMP.

Robotics with the Boe Bot Parts Kit

If you already have a Board of Education and BASIC Stamp, you may purchase the Robotics with the Boe-Bot Parts kit, with or without the printed text *Robotics with the Boe-Bot Student Guide.*

Table E-2: Robotics with the Boe-Bot Parts & Text, #28154
Robotics with the Boe-Bot Parts only, #28124
Parts and quantities subject to change without notice

Parallax Stock Code	Description	Quantity
150-01020	1 kΩ resistor	2
150-01030	10 kΩ resistor	2
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
350-00009	Photoresistors (EG&G Vactec VT935G group B)	2
350-00014	Infrared receiver (Panasonic PNA4602M or equivalent)	2
350-90000	LED standoff for infrared LED	2
350-00001	LED light shield for infrared LED	2
451-00303	3-Pin Header	2
700-00056	Whisker wire	2
700-00015	#4 screw-size nylon washer	2
710-00007	7/8” 4-40 pan-head screw, Phillips	2
713-00007	½” Spacer, aluminum, #4 round	2
800-00016	Jumper wires (bag of 10)	2
900-00001	Piezospeaker	1
28133	Boe-Bot Hardware Pack	1

Building a Boe-Bot with a HomeWork Board

If you already have a BASIC Stamp HomeWork Board that you wish to use with a Boe Bot, you will need 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

Boe-Bot Hardware Pack

All of the Boe-Bot hardware parts can be purchased individually, as found in our on-line Robot Component Shop if you find you need a replacement part. Please note that the Hardware Pack is not sold as a unit separately from the Boe-Bot Robot (Full) Kit or the Boe-Bot Parts Kit.

Table E-3: Boe-Bot Hardware Pack (#28133)
Parts and quantities subject to change without notice

Parallax Stock Code	Description	Quantity
700-00002	4-40 x 3/8”machine screw, Phillips	8
700-00003	Hex nut, 4-40 zinc plated	10
700-00009	Tail wheel ball	1
700-00016	4-40 x 3/8” flathead machine screw, Phillips	2
700-00022	Boe-Bot aluminum chassis	1
700-00023	1/16" x 1.5” long cotter pin	1
700-00025	13/32" rubber grommet	2
700-00028	4-40 x 1/ 4” machine screw, Phillips	8
700-00038	Battery holder with cable and barrel plug	1
700-00060	Standoff, threaded aluminum, round 4-40	4
721-00001	Parallax plastic wheel	2
721-00002	Rubber band tire	4
900-00008	Parallax Continuous Rotation Servo	2

Board of Education Kits

Almost all of the titles in the Stamps in Class curriculum feature different hardware and component packages that depend on the BASIC Stamp and Board of Education as a core. The Board of Education can be purchased separately or in its own kit, as listed in Table E-4 below.

Table E-4: Board of Education – Full Kit (#28102)
Parts and quantities subject to change without notice

Parallax Stock Code	Description	Quantity
28150	Board of Education Rev C	1
800-00016	Jumper wires – pack of 10	1
BS2-IC	BASIC Stamp 2 microcontroller module	1
750-00008	300 mA 9 VDC power supply	1
800-00003	Serial cable	1

Appendix F: Balancing Photoresistors

In this appendix, you will test the photoresistors to find out if they respond similarly to the same incident light levels. If the measurements they report are different for the same incident light levels, you can modify your programs to scale the values reported by your photoresistors. The values will then be similar for similar incident light levels, which can help the Boe-Bot recognize different incident light levels more reliably. This technique can in turn assist the Boe-Bot in exiting dark rooms, even with mismatched photoresistor circuits.

RC circuits with photoresistors can report different values for the same light level for many reasons: The stated value of the capacitors is 0.01 µF, but the actual value of capacitors can be very different. Many common ceramic capacitors are rated with a tolerance of +80/-20%, meaning that the actual value of the capacitor could be up to 80% larger or 20% smaller than 0.01 µF. This means that your measured decay time could also be between 80% larger and 20% smaller. The photoresistors themselves can also behave differently if they come from different manufacturing batches or if they have smudged or chipped light collecting surfaces.

F

Testing for Well Matched Photoresistor Circuits

This next example program displays the decay time of both photoresistors in the Debug Terminal. It makes it easy to judge the differences between the two readings for similar light levels.

For best results, eliminate sources of direct sunlight: In general, uniform lighting conditions improve the Boe-Bot's performance with photoresistors. Draw the blinds to eliminate sources of direct sunlight. Rooms with distributed light sources such as fluorescent lights or ceiling lamps work well.

Example Program: TestPhotoresistors.bs2

- √ Enter, save, and run TestPhotoresistors.bs2.
- √ Cast a shadow over the Boe-Bot's photoresistors with a white sheet of paper. Find a level of shade that gives you readings between 20 and 100.
- √ Record the values of both time measurements in the first row of Table F-1.

- √ Cup your hand over the photoresistors, making sure that you are casting equal shade over both. For best results, the measurements should be in the 200 to 400 range.
- √ Record the values of both time measurements in the second row of Table F-1.

Table F-1: RC-Time Measurements in Ambient and Low Light

Duration Values		Description
timeLeft	**timeRight**	
		Photoresistors in uniform ambient light
		Photoresistors in uniform low light

```
' Robotics with the Boe-Bot - TestPhotoresistors.bs2
' Test Boe-Bot photoresistor circuits.

' {$STAMP BS2}                                 ' Stamp directive.
' {$PBASIC 2.5}                                ' PBASIC directive.

timeLeft      VAR     Word                    ' Variable declarations.
timeRight     VAR     Word

DEBUG "PHOTORESISTOR VALUES", CR,             ' Initialization.
      "timeLeft  timeRight", CR,
      "--------  ---------"

DO                                            ' Main routine.

  HIGH 6                                      ' Left RC time measurement.
  PAUSE 3
  RCTIME 6,1,timeLeft

  HIGH 3                                      ' Right RC time measurement.
  PAUSE 3
  RCTIME 3,1,timeRight

  DEBUG CRSRXY, 0, 3,                         ' Display measurements.
        DEC5 timeLeft,
        "     ",
        DEC5 timeRight

  PAUSE 200

LOOP
```

Calibrating Using a Linear Approximation

The photoresistor is often referred to as a non-linear device. In other words, if it returns one measurement at one brightness, that doesn't mean that the measurement will be five times as large when the light is five times as bright. The math is more complicated and involves logarithms. However, in cases where the measurements are confined over a narrow range of the sensors overall detection abilities, the sensor can be treated like it's a linear device. You can take a couple of measurements, and then figure out how the device will react if other measurements in its range could be plotted in a straight line. The technique is called linear approximation.

Another thing you can do with linear devices is assume the difference between the two lines can also be plotted as a line. In fact, if you have one linear device that has larger measurements than the other for ambient and low light, you can use a linear approximation for making the sensors return approximately the same values for the same light levels. For every reading from one sensor, (we'll call that one x), you can multiply it by a scale factor (m), and add it to a constant (b) to get a value in the same range the other sensor would report (y).

$$y = mx + b$$

Here is an example of how to get the values of m and b to match the left photoresistor circuit to the right. First, assign X_1 and X_2 to the left photoresistor values and Y_1 and Y_2 to the right photoresistor values. (Table F-2) shows some example values for mismatched photoresistors. Your values will be different.

Table F-2: RC-Time Measurements in Ambient and Low Light

Duration Values		**Description**
timeLeft	**timeRight**	
$X_1 = 36$	$Y_1 = 56$	Photoresistors in uniform ambient light
$X_2 = 152$	$Y_2 = 215$	Photoresistors in uniform low light

Now, solve for m and b using two equations in two unknowns. One of the simpler approaches is to write two y = mx + b equations, one in terms of X_1 and Y_1 and the other in terms of X_2 and Y_2. Then, subtract one from the other to eliminate b. Then, solve for m.

$$y_2 = mx_2 + b$$
$$-(y_1 = mx_1 + b)$$
$$(y_2 - y_1) = m(x_2 - x_1)$$

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)}$$

Once you've solved for m, you can plug m back into either of the two y = mx + b equations you started with to get b.

$$y_2 = mx_2 + b$$
$$b = y_2 - mx_2$$

So, the two equations for solving for m and b turn out to be:

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \quad \text{and} \quad b = y_2 - mx_2$$

Let's plug our sample values from Table F-2 into the equations and see what the scale factor (m) and offset constant (b) will be for the left photoresistor. First, calculate m:

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)}$$
$$m = \frac{(215 - 56)}{(152 - 36)}$$
$$m = \frac{159}{116} = 1.37$$

Then, use m, y_2, and x_2 to calculate b:

$$b = 215 - (1.37 \times 152)$$
$$b = 6.76$$
$$b \approx 7$$

Now, we know how to correct the **timeLeft** variable so that it reports values similar to the **timeRight** variable in this narrow range of light levels:

$$y = mx + b$$
$$y = 1.37x + 7$$
$$timeLeft_{(adjusted)} = 1.37 \times timeLeft + 7$$

A Linear Equation in PBASIC

In most programming languages for PCs, this equation could be entered as-is. The BASIC Stamp is a very tiny processor compared to a PC. Because of this, it takes an extra step to multiply by a fractional value. You have to use the */ operator (it's called the "star-slash" operator). For the **timeLeft** equation, the PBASIC code to adjust the **timeLeft** variable can be done like this:

```
timeLeft = (timeLeft */ 351) + 7
```

The adjusted value of **timeLeft** after this line of code is executed is 1.37 times the old **timeLeft**, plus 7.

Why did 1.37 become 351? The way the */ operator works is that you have multiply your fractional value by 256, and place it to the right of the */ operator. Since 1.37 X 256 = 350.72 ≈ 351, the value 351 goes to the right of the */ operator.

You can find out more about the */ operator in the BASIC Stamp Editor by clicking Help and selecting Index. Type in */ in the field labeled "Type in keyword to find". You can also look up */ in the Binary operators section of the *BASIC Stamp Manual*.□

Your Turn – Balance Your Photoresistors with m and b

- √ In Table F-1, label the first **timeLeft** entry X1 and the second **timeLeft** entry X2.
- √ Label the first **timeRight** entry Y1 and the second **timeRight** entry Y2.
- √ Use these equations and your X1, X2, Y1, and Y2 values to solve for m and b.
- √

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \quad \textbf{and} \quad b = y_2 - mx_2$$

- √ Calculate the values of m you will use with the */ operator by multiplying m by 256.
- √ Substitute your value of m and b in this line of code from BalancePhtoresistors.bs2:

```
timeLeft = (timeLeft */ 351) + 7
```

- √ Enter, save, and run your adjusted version of BalancePhotoresistors.bs2.
- √ Expose both photoresistors to the same light level.
- √ Verify that the "after" values are similar and corrected for differences in the "before" values.
- √ Choose a different light level and again, expose both photoresistors to it.
- √ Check the "after" values again for similarity.
- √ When you have determined your values for m and b, you can modify RoamingTowardTheLight.bs2 by uncommenting the equation between **`GOSUB Test_Photoresistors`** and **`GOSUB Average_And_Difference`**. Your m value will replace **`351`** and your b value will replace **`7`**.

```
' Robotics with the Boe-Bot - BalancePhotoresistors.bs2
' Test adjustments to Boe-Bot photoresistor circuits.

' {$STAMP BS2}                                ' Stamp directive.
' {$PBASIC 2.5}                               ' PBASIC directive.

timeLeft      VAR     Word                   ' Variable declarations.
timeRight     VAR     Word

DEBUG "PHOTORESISTOR VALUES", CR,            ' Initialization.
      "timeLeft  timeRight", CR,
      "--------  ---------"

DO                                           ' Main routine.

  HIGH 6                                     ' Left RC time measurement.
  PAUSE 3
  RCTIME 6,1,timeLeft

  HIGH 3                                     ' Right RC time measurement.
  PAUSE 3
  RCTIME 3,1,timeRight


  DEBUG CRSRXY, 0, 3,                        ' Display measurements.
        DEC5 timeLeft,
        "     ",
```

```
        DEC5 timeRight,
        "   Before"

  timeLeft = (timeLeft */ 351) + 7

  DEBUG CRSRXY, 0, 5,                          ' Display measurements.
        DEC5 timeLeft,
        "     ",
        DEC5 timeRight,
        "   After"

  PAUSE 200

LOOP
```

Appendix G: Tuning IR Distance Detection

Finding the Right Frequency Sweep Values

Fine tuning the Boe-Bot's distance detection involves determining which frequency is most reliable for each zone for each IR pair.

Note: This appendix features a method of determining the best frequencies for determining given distances using spreadsheets. This activity takes time and patience, and is only recommended if your distance sensing is severely out of calibration. It involves collecting frequency sweep data and using it to determine the most reliable values for detecting particular distances.□

- √ Point both the IR LEDs and detectors straight forward.
- √ Place the Boe-Bot in front of a wall with a white sheet of paper as the IR target.
- √ Place the Boe-Bot so that its IR LEDs are 2.5 cm away from the paper target. Make sure the front of the Boe-Bot is facing the paper target. Both IR LEDs and detectors should be pointed directly at the paper.

IR Fine Tuning Program

FrequencySweep.bs2 performs a frequency sweep on the IR detector and displays the data. Although the techniques used are similar to other programs, it has one unique feature. The BASIC Stamp is programmed to wait for you to press the Enter key.

- √ Enter and run FrequencySweep.bs2, but do not disconnect the Boe-Bot from the serial cable.

```
' -----[ Title ]-----------------------------------------------------------
' Robotics with the Boe-Bot - FrequencySweep.bs2
' Test IR LED/detector response to frequency sweep.

' {$STAMP BS2}                              ' Stamp directive.
' {$PBASIC 2.5}                             ' PBASIC directive.

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

crsrPosRow     VAR     Byte
irFrequency    VAR     Word
irDetect       VAR     Bit
distance       VAR     Nib
dummy          VAR     crsrPosRow
```

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

DEBUG CLS,
      "Click transmit windowpane,", CR,
      "then press enter to begin", CR,
      "frequency sweep...", CR, CR,

      "           OBJECT", CR,
      "FREQUENCY  DETECTED", CR,
      "---------  --------", CR

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

DO

  DEBUGIN dummy

  crsrPosRow = 6

  FOR irFrequency = 30500 TO 46500 STEP 1000

    crsrPosRow = crsrPosRow + 1

    FREQOUT 8,1, irFrequency
    irDetect = IN9

    DEBUG CRSRXY, 4, crsrPosRow, DEC5 irFrequency
    DEBUG CRSRXY, 11, crsrPosRow

    IF (irDetect = 0) THEN DEBUG "Yes" ELSE DEBUG "No "

    PAUSE 100

  NEXT

LOOP
```

- √ Click the upper of the two window panes shown in Figure G-1.
- √ Press the Enter key. The frequency response data will appear as shown in the figure.

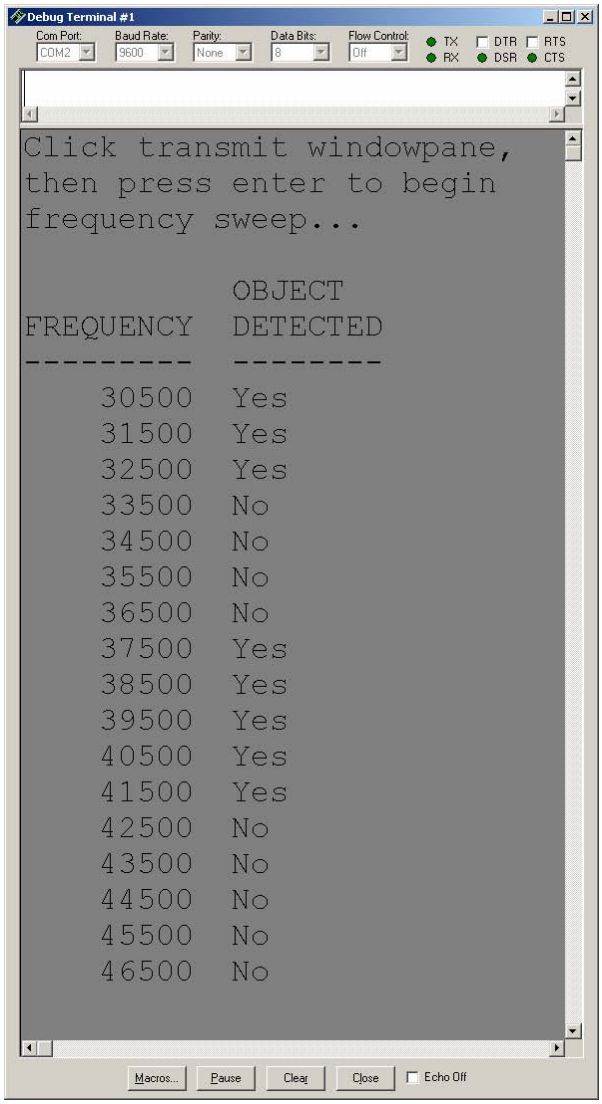


Figure G-1
Debug of Frequency Data

The BASIC Stamp has been programmed to make the Debug Terminal display a "Yes" if an object was detected and a "No" if an object was not detected. Figure G-1 shows that the left sensor's region of good signal response is between 36500 and 42500.

- √ Modify the **FOR**...**NEXT** loop in Program Listing FrequencySweep.bs2 so that it steps in increments of 250 and includes the upper and lower limits of both detectors. Based on the data in the example shown in Figure G-1, the start, end, and step values of the **FOR**...**NEXT** loop would be modified as follows:

```
FOR irFrequency = 36500 to 42500 STEP 250
```

- √ Re-run your modified FrequencySweep.bs2, and press Enter again.
- √ Record the data for left and right sides in separate spreadsheets.
- √ Press the Enter key again and record the next set of data points.
- √ Repeat this process three more times. When finished, you will have five sets of data points for each sensor in separate spreadsheets for this one frequency.
- √ Back the Boe-Bot up 2.5 cm. Now your Boe-Bot's IR detectors will be 5 cm from the paper target.
- √ Record five more data sets at this distance.
- √ Keep on backing up the Boe-Bot by 2.5 cm at a time and recording the five frequency sweep data sets between each distance adjustment.
- √ When the Boe-Bot has been backed up by 20 cm, the frequency sweep will display mostly, if not all "No" regions. When the frequency sweep is all "No", it means no object is detected at any frequency within the sweep.

By careful scrutiny of the spreadsheets and process of elimination, you can determine the optimum frequency for each IR pair for each zone. Customizing for up to eight zones can be done without any restructuring of the Boe-Bot navigational routines. If you were to customize for 15 zones, this would entail 30 one millisecond **FREQOUT** commands. That won't gracefully fit between servo pulses. One solution would be to take 15 measurements every other pulse.

How to determine the best frequencies for the left sensor is discussed here. Keep in mind you'll have to repeat this process for the right sensor. This example assumes you are looking for six zones (zero through five).

- √ Start by examining the data points taken when the Boe-Bot was furthest from the paper target. There probably won't be any sets of data points that are all "Yes" readings at the same frequency. Check the data points for the next 2.5 cm towards the paper target. Presumably, you will see a set of four or five "Yes"

readings at a particular frequency. Note this frequency as a reliable measurement for the dividing line between Zone 0 and Zone 1.

√ At each of the remaining five distances, find a frequency for which the output values have just become stable.

For example, at 15 cm, three different frequencies might show five ”Yes” readings. If you look back to the 17.5 cm mark, two of these frequencies were stable, but the other was not. Take the frequency that was not stable at 17.5 cm but was stable at 15 cm as your most reliable frequency for this distance. Now, this example has determined the frequencies that can be used to separate Zones 5 and 4 and Zones 4 and 3. Repeat this process for the remaining zone partitions.

Your Turn

√ If you succeeded in fine tuning five measurements and time permits, try increasing the resolution to eight measurements. Save your data for both methods.

Appendix H: 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.

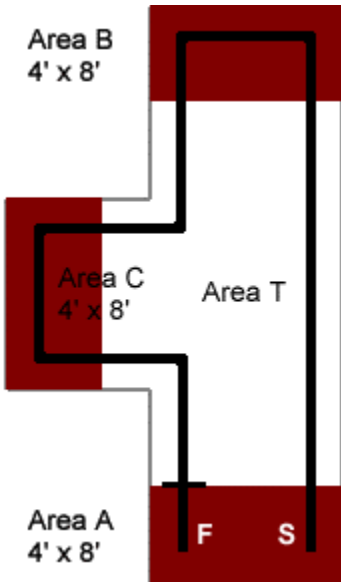


Figure H-1
Sample Contest Course

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.

Table H-1: 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.

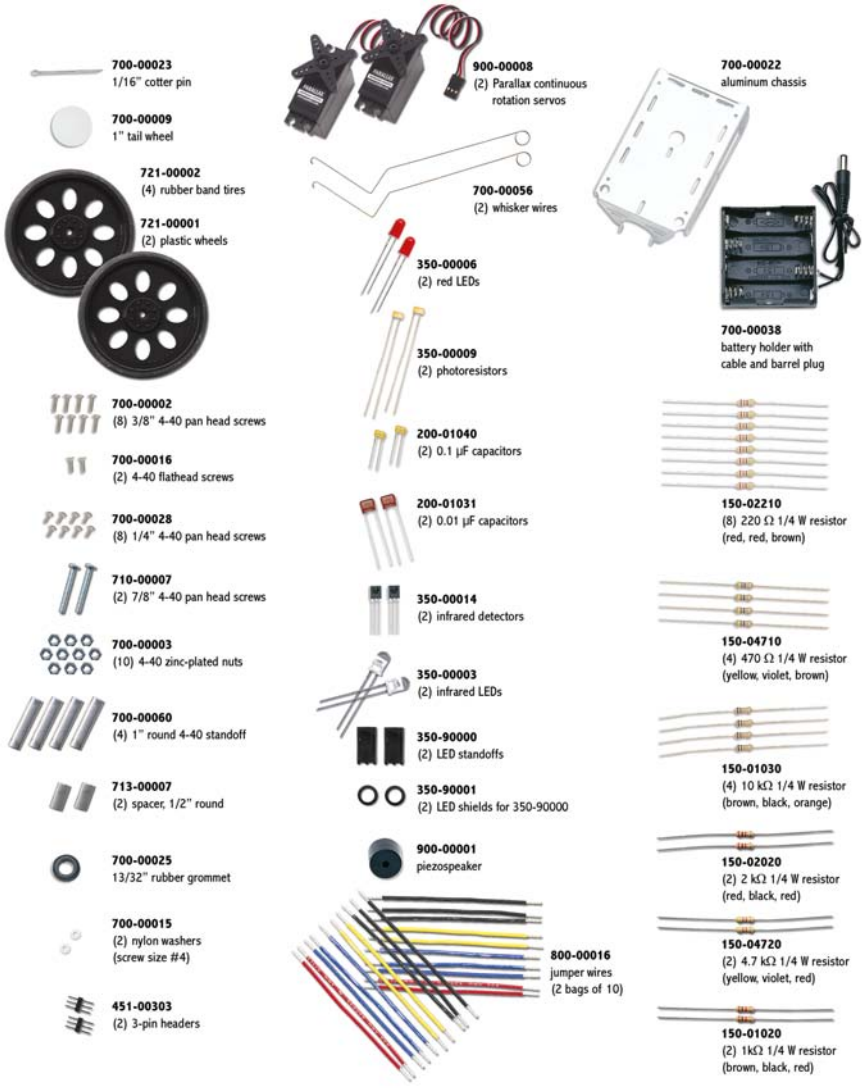
Scoring

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 in the various Boe-Bot Robot kits are subject to change without notice. Parts may differ from what is shown in this picture. Please contact stampsinclsss@parallax.com if you have any questions about your kit.