Robotics!

Student Workbook

Version 1.5

Note regarding the accuracy of this text:
Accurate content is of the utmost importance to the authors and editors of the Stamps in Class texts. If you find any error or subject that needs clarification, please report it to stampsinclass@parallaxinc.com.

PARALLAX INC

Warranty

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

14-Day Money Back Guarantee

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

Copyrights and Trademarks

This documentation is copyright 2001 by Parallax, Inc. BASIC Stamp is a registered trademark of Parallax, Inc. If you decide to use the name BASIC Stamp on your web page or in printed material, you must state: "BASIC Stamp is a registered trademark of Parallax, Inc." Other brand and product names are trademarks or registered trademarks of their respective holders.

Disclaimer of Liability

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

Internet Access

We maintain Internet systems for your use. They can be used to obtain free Parallax software and documentation and also to purchase Parallax products. These systems may also be used to communicate with members of Parallax and other customers. Access information is shown below:

E-mail: stampsinclass@parallaxinc.com
Web: http://www.parallaxinc.com and http://www.stampsinclass.com

Internet BASIC Stamp Discussion Lists

We maintain two e-mail discussion lists for people interested in BASIC Stamps (subscribe at http://www.parallaxinc.com under the technical support section). The BASIC Stamp list server includes engineers, hobbyists, and enthusiasts. The list works like this: lots of people subscribe to the list, and then all questions and answers sent to the list are distributed to all subscribers. It's a fun, fast, and free way to discuss BASIC Stamp issues and get answers to technical questions. This list generates about 40 messages per day.

The Stamps in Class list is for students and educators who wish to share educational ideas (subscribe at http://www.stampsinclass.com under the discuss/e-mail section). This list works the same way the BASIC Stamp list server does, and it currently generates about five messages per day.

Table of Contents

Preface v

Audience and Teacher's Guides vi
Copyright and Reproduction vi
Typographical Conventions vii
Robotics! Contributors viii

Read this First - Before You Start 1

Check Your Servo Labels 1
Use the Right Power Supply 2
The New Stamps in Class Robotics! Web Page 2

Chapter #1: Assembling and Testing Your Boe-Bot 5

About Robotics Competitions and Robot Development 5
Activity #1: Boe-Bot Parts and Tools 6
Activity #2: Boe-Bot Mechanical Assembly 9
Activity #3: Programming The Boe-Bot's BASIC Stamp 2 On-Board Computer 19
Activity #4: Testing the Servos Individually 26
Activity #5: Running Both Servos 33
Activity #6: Tuning the Servos – Calibration in Software 35
Summary and Applications 38
Questions and Projects 40

Chapter #2: Programming the Boe-Bot to Go Places 43

Converting Instructions to Motion 43
Activity #1: Low Battery Indicator 44
Activity #2: Controlling Distance 48
Activity #3: Maneuvers – Making Turns 53
Activity #4: Maneuvers – Ramping 55
Activity #5: Remembering Long Lists Using EEPROM 57
Activity #6: Simplify Navigation with Subroutines 62
Activity #7: All Together Now 64
Summary and Applications 70
Questions and Projects 71

Chapter #3: Tactile Navigation with Whiskers 75

Tactile Navigation 75

Activity #1: Building and Testing the Whiskers 75
Activity #2: Navigation With Whiskers 82
Activity #3: Looking at Multiple Inputs as Binary Numbers 86
Activity #4: Artificial Intelligence and Deciding When You're Stuck 90
Summary and Applications 95
Questions and Projects 96

Chapter #4: Light Sensitive Navigation with Photoresistors 99

Is Your Boe-Bot a Photophile or a Photophobe? 99
Activity #1: Building and Testing Photosensitive Eyes 100
Activity #2: A Light Compass 104
Activity #3: Follow the Light! 107
Activity #4: Line Following 110
Summary and Applications 114
Questions and Projects 115

Chapter #5: Object Detection Using Infrared 117

Using Infrared Headlights to See the Road 117
Infrared Headlights 117
The Freqout Trick 118
Activity #1: Building and Testing the New IR Transmitter/Detector 119
Activity #2: Object Detection and Avoidance 123
Activity #3: Navigating by the Numbers in Real-Time 126
Summary and Applications 130
Questions and Projects 131

Chapter #6: Determining Distance Using Frequency Sweep 133

What's a Frequency Sweep? 133
Activity #1: Testing the Frequency Sweep 133
Activity #2: The Drop-off Detector 140
Activity #3: Boe-Bot Shadow Vehicle 145
Summary and Applications 151
Questions and Projects 153

Appendix A: Boe-Bot Parts Lists and Sources 155

Appendix B: PC to Stamp Communication Trouble-Shooting 159

Appendix C: PBASIC Quick Reference 161

Appendix D: Building Servo Ports on the Rev A Board of Education 169

Appendix E: Board of Education Rev A Voltage Regulator Upgrade Kit 173

Appendix F: Breadboarding Rules 175

Appendix G: Resistor Color Codes 177

Appendix H: Tuning IR Distance Detection 179

Appendix I: Boe-Bot Competition Maze Rules 185

Preface

Robots are used in the auto, medical, and manufacturing industries, and of course, in many science fiction films. Building and programming a robot is a combination of mechanics, electronics, and problem solving. What you're about to experience with the Boe-Bot will be relevant to realistic applications using robotic control, the only difference being the size and sophistication. The electronic control principles, example program listings, and circuits you will use are very similar (and sometimes identical) to industrial applications developed by engineers.

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 it onward, it seemed that many science fiction stories involved these robots revolting against human authority. 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.

This series of experiments will introduce you to basic robotic concepts using the Board of Education Robot (hereafter the "Boe-Bot"). The experiments will begin with construction of the Boe-Bot. After that, we'll program the Boe-Bot for basic maneuvers, and proceed to add sensors that will allow it to react to its surroundings. The goal of this text is to show students how easy it is to become interested in and excited about the fields of engineering, mechatronics, and software development as they design, construct and program an autonomous robot. The Boe-Bot provides students with a project area to build and customize their own mechanical, electrical, and programming projects. The use of a Boe-Bot to introduce microcontroller circuits and interfacing is ideal since the outputs are almost entirely visible and easy to customize.

The Board of Education Rev B, which serves as the Boe-Bot’s prototyping platform, was designed for use with all five Stamps in Class series of experiments, including Robotics! The Board of Education, Rev B has four servo ports, and this makes it possible to use four servos without taking up any space on the breadboard prototyping area. Each port has a dedicated I/O line (P12, P13, P14, or P15 depending on the port), and each can be used for controlling a servo. Each servo port supply is tied to Vin, the unregulated 6 V supply from the battery pack, so use of a higher voltage supply is discouraged due to its tendency to overwork the servos. The Board of Education Rev B also has two large capacitors that stabilize the BASIC Stamp’s power supply. They ensure that the BASIC Stamp operates continuously, even when the servos are performing direction changes, which could otherwise cause brownout conditions.

The Robotics curriculum is periodically revised and updated based on feedback from students and educators. If you would like to author an addition to this curriculum, or have ideas for improvements, please send them to stampsinclass@parallaxinc.com. We'll do our best to integrate your ideas and assist you with whatever technical support, sales support, or on-site training you need. If we accept your Boe-Bot project, we'll send you a free Boe-Bot.

Audience and Teacher's Guide

The Robotics curriculum was created for ages 15+ as a subsequent text to the "What's a Microcontroller?" guide. Like all 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 based on 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.

Experts in their field independently author each set of Stamps in Class experiments, and they are provided leeway in terms of format. As a result, the depth and availability of teachers' guides varies. Please contact Parallax, Inc. if you have any questions. If you are interested in contributing material to the Stamps in Class series, please submit your proposal to stampsinclass@parallaxinc.com.

Copyright and Reproduction

Stamps in Class curriculum is copyright © Parallax 2001. Parallax grants every person conditional rights to download, duplicate, and distribute this text without our permission. The condition is that this text or any portion thereof, should not be duplicated for commercial use resulting in expenses to the user beyond the marginal cost of printing. Preferably, duplication would have no expense to the student. Any educational institution wishing to produce duplicates for its students may do so without our permission. This text is available in printed format from Parallax. Because we print the text in volume, the consumer price is often less than typical xerographic duplication charges. This text is also available for free download from the www.stampsinclass.com -> Downloads -> Educational Curriculum page in PDF format. Documents in this format can be viewed and printed using Adobe Systems' Acrobat® Reader software available from www.adobe.com. This software can also be installed directly from the Parallax CD.

This text may be translated to any other language with prior permission of Parallax, Inc.

Typographical Conventions

- Checklist instruction. The square box indicates a "how to" instruction. These instructions should be followed sequentially, like a checklist, through each activity in this text.

✓ **TIP**	Pay attention to and follow these instructions. They will make the activities easier and save time.

FYI	This box contains useful information.

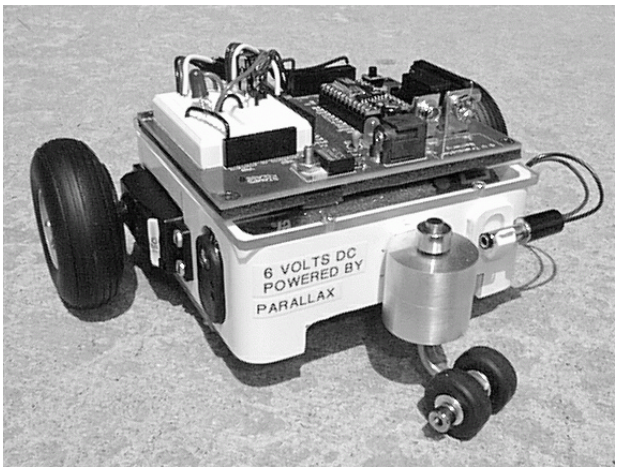
⚠	**Caution: follow these instructions, or you may end up damaging your hardware.**

```
' PBASIC Program Listings.
```

```
' PBASIC excerpt from a program listing.  This kind of excerpt
' always follows a paragraph of text explaining what it does
' and how it works.
```

PBASIC code in a paragraph of text takes the form of: **`command`** ***`argument1, argument2, etc`***. Note that the command is not italicized, but its arguments are.

Robotics! 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 here along with many 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.

Andrew Lindsay, Parallax Chief Roboticist, wrote the majority of the v1.3 text with three goals in mind. First, support all activities in the text with carefully written "how to" instructions. Second, expose the reader and student to new circuit, programming, engineering and robotic concepts in each chapter. Third, ensure that the experiments can be performed with a high degree of success using either the Rev A or Rev B Board of Education. Parallax 2000 summer intern, Branden Gunn, assisted in the illustration of this revision.

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.

If you have suggestions, think you found a mistake, or would like to contribute an activity or chapter to forthcoming Robotics! v1.6 or More Robotics! texts, contact us at stampsinclass@parallaxinc.com. Subscribe and stay tuned to the Stamps in Class e-group for the latest in free hardware offers for Robotics! contributions. See the Internet BASIC Stamp Discussion Lists section just before the Table of Contents for information on how to subscribe.

Read this First - Before You Start

This Robotics! update contains three important messages:

1. Check your Servo Labels
2. Use the Right Power Supply
3. New Robotics! Web Site

Check Your Servo Labels

Starting in June, 2001, Parallax will ship all Robotics! kits with pre-modified servos. The Robotics! v1.5 student workbook is written exclusively for use with Boe-Bots that have pre-modified servos. Pre-modified servos are labeled "PM". If you have a Boe-Bot purchased before June, 2001, it most likely has standard servos, which are labeled "STD". If you have Standard servos, use the Robotics! v1.4 text. Both versions of the Robotics! Student Workbook (v1.4 and v1.5) are available for free download from the www.stampsinclass.com -> Robotics page.

If you have questions about whether your servos are pre-modified or standard, check the label on the front of each servo against those shown in the Servo Identification Table below.

Parallax Servo	Servo Identification Table Check the labeling on the servos in your Robotics! kit.	Examples of the labeling for pre-modified (PM) and standard (STD) servos
	Use Robotics! v1.5 (this text) Use this student workbook only if the letters PM are shaded on the label on the front of your servos.	PARALLAX STD PM BB HS HT www.parallaxinc.com
	Use Robotics! v1.4 If the letters PM are not shaded or do not appear on your servo's labeling, use the Robotics v.1.4 Student Workbook available for free download from the www.stampsinclass.com - > Robotics page.	PARALLAX ■ STD □ BB □ HS www.parallaxinc.com

Read this First – Before You Start

Use the Right Power Supply

The Boe-Bot is designed for use with the battery pack that comes with the Robotics! kits. When selecting batteries for the Boe-Bot:

- Use only AA 1.5 V batteries with this battery pack.
- Do not use 1.2 V rechargeable AA batteries.

Do not use a 9 V battery or AC adaptor; it could damage your Boe-Bot's servo motors.

If you want to use a wall mount AC adaptor and save batteries for autonomous navigation, make sure your AC adaptor has these output specifications (preferred values are **bold**):

Output:

- Voltage rating should be **6 V DC** or 7.5 V DC
- Current rating from 600 mA to **1000mA (1 A)**
- 2.1 mm center positive barrel plug

Make sure the AC adaptor's label has the center positive symbol

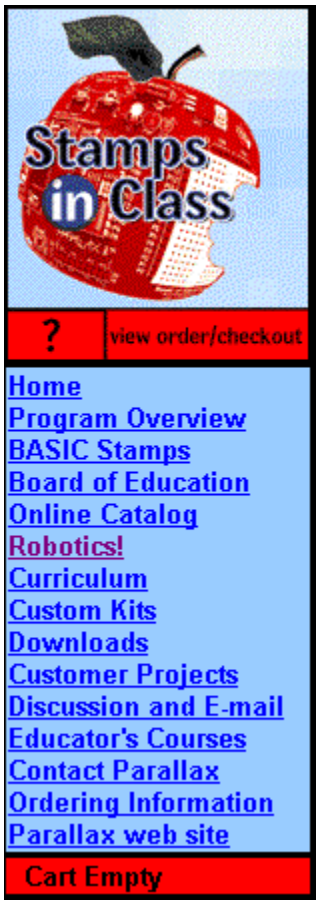
The New Stamps in Class Robotics! Web Page

Visit www.stampsininclass.com -> Robotics (see facing page). This page contains:

- Student project examples using the Boe-Bot
- More Boe-Bot activities for students after they have completed the Robotics! Student Workbook
- Boe-Bot application kits
- Boe-Bot application modules

Students and instructors are encouraged to submit projects to stampsinclass@parallaxinc.com for posting to this resource site. Hobbyists and hardware developers are also encouraged to submit proposals, proofs of concept, or completed and documented Boe-Bot application kits/add-on modules.

From the www.stampsininclass.com -> Robotics Web Page



Coming Soon - Boe-Bot Application Kits and Modules:

Kit - IR Wheel Encoder

Module – Line Follower
Module - Compass

Robotics! is our most popular series. With over 10,000 Parallax Boe-Bots in use around the world, it's clearly the light, sound and movement that immediately captures the interest of the student. This is not a toy; the concepts are directly applicable to microcontroller interfacing and code development.

Follow these links to see:

Student Project Examples **More Boe-Bot Activities**

Boe-Bot Application Kits **Boe-Bot Application Modules**

Student Projects:

Title	Author	Overview
Maze Runners	9th Grader, Laura Wong	Includes introductions to the mechanical problems associated with maze navigation, state machine design for maze navigation, and PBASIC program examples used with the Boe-Bot. Parallax grade: A+, Great work Laura!

Boe-Bot Applications Notes:

Title	Concepts Introduced/Covered
Controlling Your Boe-Bot with a Universal Remote	Includes introduction to pulse width modulation for communication, examples (with PBASIC programming examples for the Boe-Bot) of reading IR remote control codes, sending codes to your Boe-Bot and controlling your Boe-Bot using the Channel and Volume keys.
On/Off with Reset	Use the Reset button on the Boe-Bot to toggle Program Execution on/ff.

Chapter #1: Assembling and Testing Your Boe-Bot

About Robotics Competitions and Robot Development

Students in high schools and colleges preparing their entries for various robotics competitions get first-hand exposure to the engineering occupation. They start by working in teams developing a Robot's subsystems. A robot's subsystems include its motors, sensor arrays, microprocessor, and mechanical linkages. Next they test and trouble-shoot the subsystems. Then comes system integration, the process of making all the Robot's subsystems work together.

Once the testing and trouble-shooting is finished at the subsystem level, a robot's subsystems have to be connected to and controlled by a microprocessor. The process of getting all the subsystems (including the microprocessor) to work together to make the robot perform its assigned task list is called system integration. System integration can be tricky to begin with, but robotics teams who skipped any of the testing and troubleshooting at the subsystem level often have much larger problems with their system integration. Many a late night can be spent trying to get the robot to work the way it's supposed to. If bugs are hiding in the subsystems when you're trying to do system integration, it only compounds the problems.

Even when testing and trouble shooting is performed for each subsystem, it can still be the most difficult part of robot development. For example, a group at a recent robotics competition spent five hours trying to get a Sumo wrestling robot to work right with no luck. Later, by utilizing the BASIC Stamp's Debug Terminal, the testing and troubleshooting took less than 5 minutes.

FYI	The term BASIC Stamp will be used throughout this text to refer to the BASIC Stamp 2.

Testing and troubleshooting at each phase of robot development is a skill that one gets better at with practice. By following the instructions in the activities in this student workbook, you'll get a taste of testing and trouble shooting while putting your Boe-Bot together and getting it up and running. With practice, you'll enjoy more five-minute troubleshooting times and less of the five-hour variety.

Chapter #1: Assembling and Testing Your Boe-Bot

This chapter is separated into six activities:

1. Boe-Bot Parts and Tools
2. Boe-Bot Mechanical Assembly
3. Programming the Boe-Bot's BASIC Stamp 2 On-Board Computer
4. Testing the Servos Individually
5. Running Both Servos
6. Tuning the Servos – Calibration in Software

Each of these activities involves discrete steps to get the Boe-Bot up and running. First, check to make sure you have all your parts. Next, put the mechanical parts together. After that, test the microprocessor subsystem. Then test each servo motor individually. Then, make the servo motors work in unison. Last, but certainly not least, calibrate the pre-modified servos. By carefully following the instructions in these first six activities, you ensure that your microprocessor and motor subsystems are working reliably. The task in later chapters will be to develop and test a variety of sensors and integrate them with the rest of the Boe-Bot's subsystems. In Chapters 3-6, you'll isolate and test the sensors before writing PBASIC programs that integrate the sensor subsystems. For example, in chapter 3, you'll first construct and test whiskers, sensors that tell the Boe-Bot when it's bumped into something. Once the testing and trouble-shooting is complete, you'll move on to writing PBASIC programs that make use of the whisker input signals for directing the Boe-Bot's motion.

Activity #1: Boe-Bot Parts and Tools

Let's get started by taking an inventory of the tools and parts we'll need to get though the activities in this student workbook. For starters, **all activities in this student workbook require a personal computer (PC) with the Windows 95/98/... operating system**. You'll also need a few simple hand tools, all of which are common and can be found in most households, and school shops. They can also be purchased at local hardware stores. The parts for the Boe-Bot are either included in the Boe-Bot full kit or in a combination of the BOE Full Kit and the Robotics! parts kit. See Appendix A: Boe-Bot Parts Lists and Sources for more information.

The Simple Hand Tools

Recommended Tools

The top row of tools in Figure 1.1 are recommended for the Activities in Chapter #1.

(1) Phillips #1 point screwdriver
(1) ¼" Combination wrench

The tools shown on the bottom row will come in handy for the activities from Chapter #2 onward.

(1) Small needle nose pliers
(1) Wire cutter/stripper

Figure 1.1: Recommended tools.

Boe-Bot Parts Inventory

- Before getting started, take an inventory of the parts in your kit. Appendix A: Boe-Bot Parts Lists and Sources will tell you how many of each part should be in your kit. For help with identifying each part, use the back cover of this text; it has labeled pictures of all of the Boe-Bot parts.

- Gather the parts shown in Figure 1.2 and set them aside for use as you go through the rest of the activities in this chapter.

Chapter #1 Parts List:

A	(1)	Boe-Bot chassis
B	(1)	Battery pack
C	(2)	Parallax Pre-Modified Servos (labeled PM)
D	(2)	Plastic wheels
E	(1)	Polyethylene ball
F	(2)	9/32” Rubber Grommets
G	(1)	13/32” Rubber Grommet
H	(1)	Board of Education and BASIC Stamp 2
I	(2)	O-ring tires
J	(1)	Cotter pin
K	(10)	4-40 locknuts
L	(2)	4-40 flathead screws
M	(8)	3/8” 4-40 screws
N	(8)	1/4” 4-40 screws
O	(4)	1/2" Standoffs
P	(1)	Serial cable
Q	(4)	AA alkaline batteries
R	(1)	Parallax CD

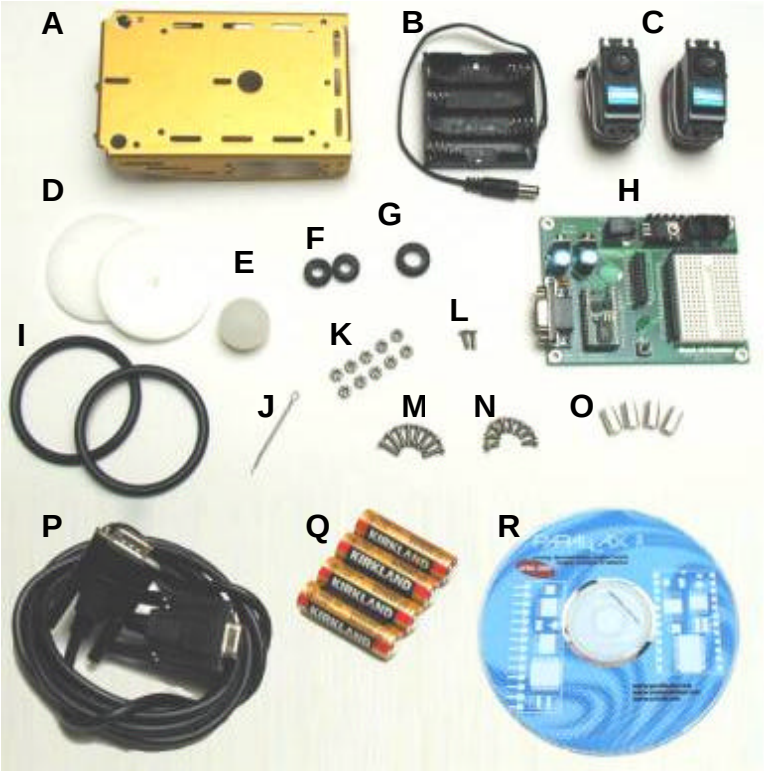


Figure 1.2: Chapter #1 parts.

Activity #2: Boe-Bot Mechanical Assembly

This section breaks assembling the Boe-Bot into steps. In each step, you 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.

Mounting the Topside Hardware

Figure 1.3 shows the Boe-Bot chassis, topside hardware and mounting screws.

Parts List:

(1) Boe-Bot Chassis
(4) Standoffs
(4) 1/4" 4-40 Screws
(2) 9/32" Rubber grommets
(1) 13/32" Rubber grommet

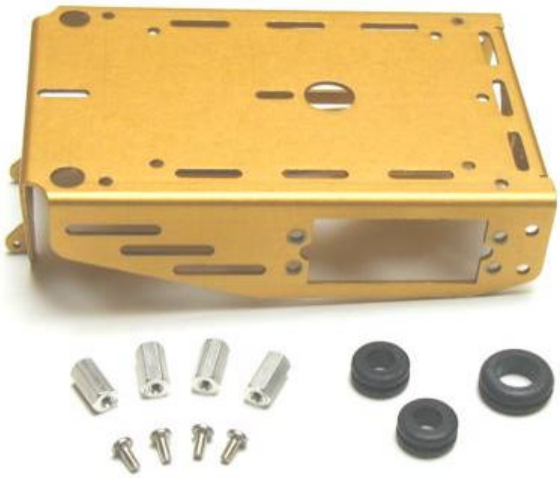


Figure 1.3: Chassis and topside hardware.

Assembly:

Figure 1.4 shows the topside hardware attached to the Boe-Bot chassis. Each rubber grommet has a groove in its outer edge that holds it in place in a hole on the top of the Boe-Bot chassis.

- Insert the 13/32" rubber grommet into the hole in the center of the Boe-Bot chassis.
- Insert the two 9/32" rubber grommets into the two corner holes as shown.
- Use the four 1/4" 4-40 screws to attach the four standoffs to the chassis as shown.

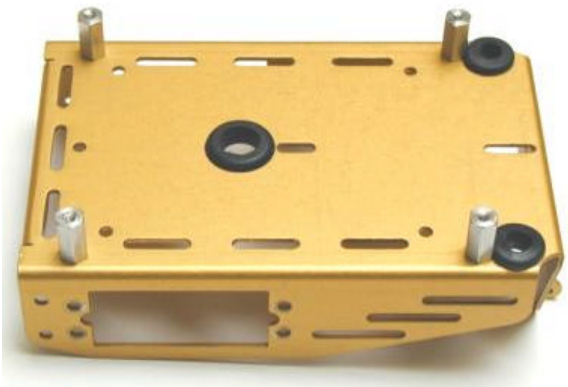


Figure 1.4: Topside hardware assembled.

Removing the Servo Horns

Get the two Parallax pre-modified servos from your parts kit, shown in Figure 1.5. Each servo has a horn attached to its output shaft by a Phillips screw.

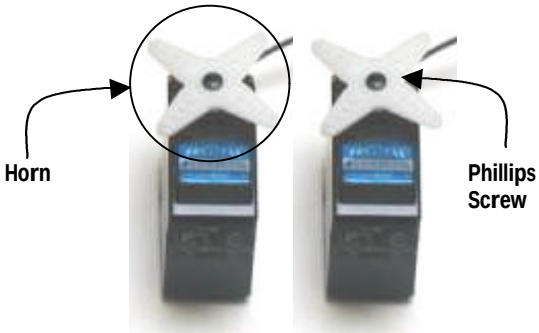


Figure 1.5: Parallax pre-modified servos.

Parts List

(2) Pre-modified servos

Figure 1.6 shows the dehorned servos.

- Unscrew each of the Phillips screws, then pull each servo horn upwards and off of the servo output shaft.
- Save the screws for attaching the Boe-Bot wheels.

Figure 1.6: Pre-modified servos dehorned.

Mounting The Servos

Parts List:

Figure 1.7 shows the pre-modified servos and servo mounting hardware.

(1) Partially assembled Boe-Bot chassis
(2) Servos
(8) 3/8" 4-40 screws
(8) 4-40 locknuts

Stop

If you have not already checked the labeling on your servos, do that now. Turn to page 1 and follow the instructions.

Figure 1.7: Servos and mounting hardware.

Assembly:

Figure 1.8 shows the servos mounted on the chassis.

- Use the eight 3/8" 4-40 screws and locknuts to attach each servo to the Boe-Bot chassis as shown.

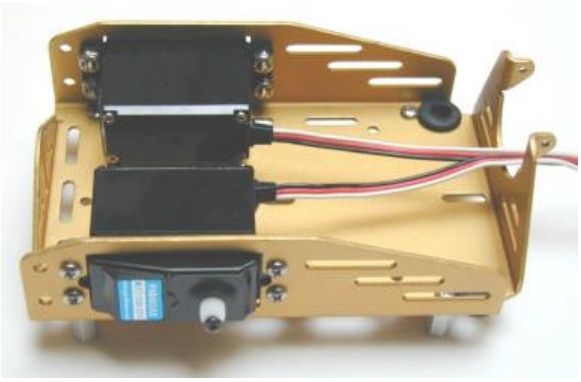


Figure 1.8: Servos mounted on chassis.

Mounting the Battery Pack

Figure 1.9 shows the battery pack and mounting hardware to be added next.

Parts List:

(1) Partially assembled Boe-Bot chassis.
(1) Empty battery pack
(2) Flathead 4-40 screws
(2) 4-40 locknuts

Figure 1.9: Battery pack and mounting hardware.

Assembly

Figure 1.10 shows the Boe-Bot chassis with the battery pack mounted (a) from the underside and (b) from the topside.

- Use the flathead screws and locknuts to attach the battery pack to underside of the Boe-Bot chassis as shown in Figure 1.10 (a). Make sure to insert the screws through the battery pack then tighten down the locknuts on the topside of the chassis.

- Pull the battery pack's power cord through the hole with the largest 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 in Figure 1.10 (b).

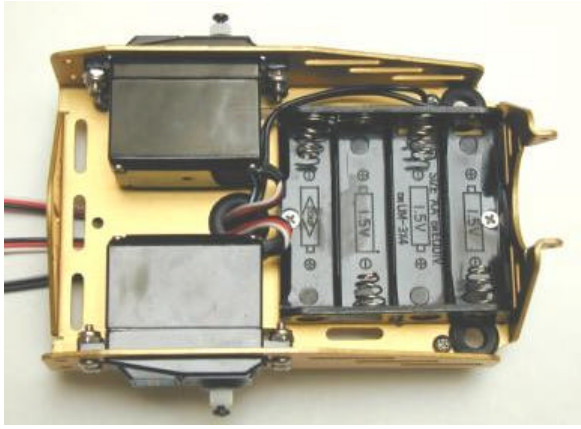
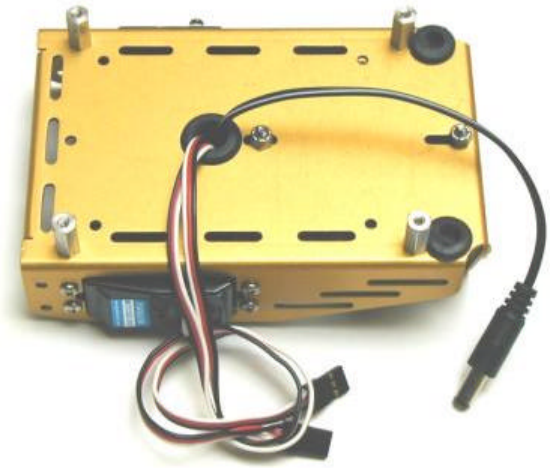


Figure 1.10: (a) Battery pack installed



(b) wires pulled through.

Socketing the BASIC Stamp 2.

Figure 1.11 shows the BASIC Stamp 2 to the left of the Board of Education Rev B.

Parts List:

(1) Basic Stamp 2
(1) Board of Education Rev B.

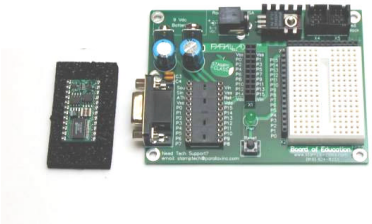


Figure 1.11: BASIC Stamp 2 and Board of Education Rev B.

FYI	Board of Education is abbreviated BOE, and Rev B stands for revision-B.

Figure 1.12 shows the BASIC Stamp 2 mounted in its socket on the BOE. The BASIC Stamp has a half-circle printed in the center of its top edge. This is meant to serve as the reference notch common on many integrated circuits. When placing the BASIC Stamp in its socket on the BOE, make sure this half-circle is closest to the Sout and Vin labels. As a second check, make sure the largest black chip with the label PIC16C57C is at the bottom, between the P7 and P8 labels.

- If your BASIC Stamp and BOE were packaged separately, plug the BASIC Stamp into its socket on the BOE as shown in Figure 1.12. Make sure the pins on the BASIC Stamp line up with the holes in the socket, then press down firmly on the BASIC Stamp with your thumb. The BASIC Stamp's pins should sink into socket holes by about a quarter-inch.

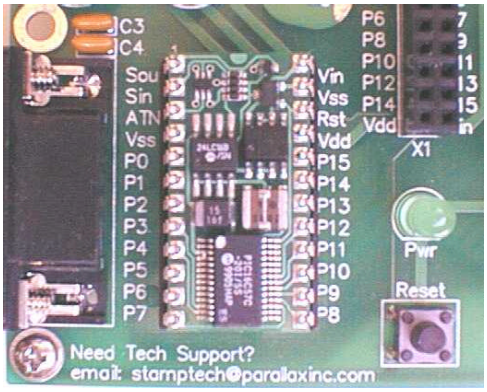


Figure 1.12: BASIC Stamp 2 inserted into its socket on the BOE.

Attaching the Board of Education to the Boe-Bot Chassis

Figure 1.13 shows the Board of Education, BASIC Stamp and mounting hardware.

<u>Parts List:</u>

(1) Partially assembled Boe-Bot (not shown)
(1) Board of Education with BASIC Stamp 2
(4) 1/4" 4-40 screws

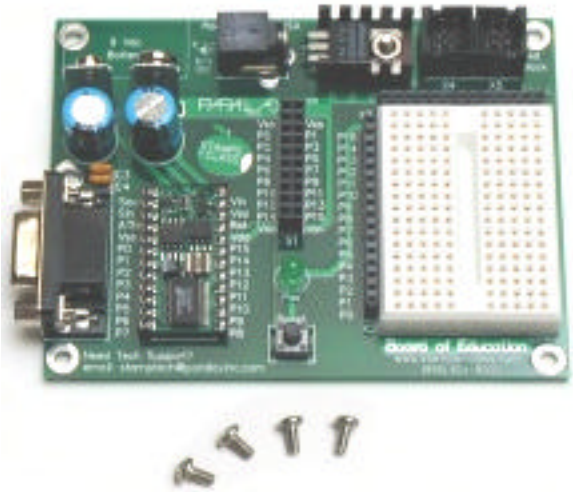


Figure 1.13: BOE with BASIC Stamp and mounting screws.

Assembly:

Figure 1.14 shows the Board of education attached to the Boe-Bot chassis with the servos plugged into the servo ports.

- Make sure the white breadboard on the Board of Education is above where the servos are mounted on the chassis.
- Use the four 1/4" machine screws to attach the Board of Education to the standoffs.

Figure 1.14: BOE attached to chassis.

Figure 1.15 (a) shows a close-up of the servo ports on the BOE Rev B. The numbers along the top indicate the servo port number. If you connect a servo to servo port 12, it means the servo's control line is connected to I/O line P12. I/O line P12 is a metal trace on the BOE that connects the top servo port pin to the BASIC Stamp's I/O pin P12.

The labels to the right of the servo port are for making sure your servo gets plugged in properly. Figure 1.15 (b) shows a servo plugged into servo port 12 so that the black wire lines up with the "Black" label, and the red wire lines up with the "Red" label. Although the topmost wire is labeled "White" in Figure 1.15 (b), it could either be white or yellow.

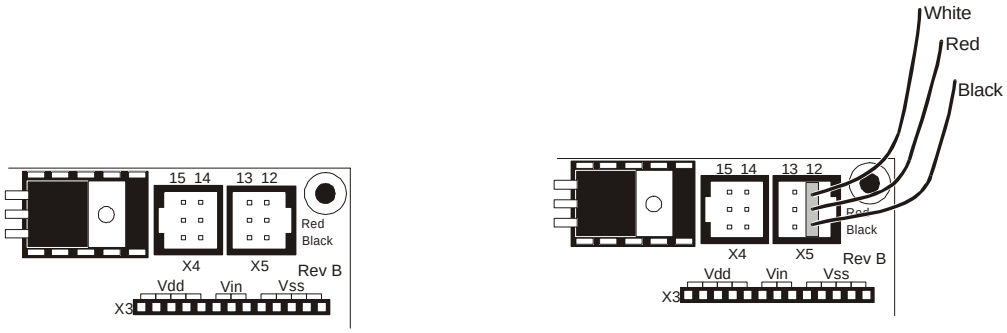


Figure 1.15: Servo ports on the BOE Rev B (a) before, and (b) after plugging in servo port 12.

Make sure the "Black" and "Red" labels to the right of the servo port line up with the servo connector's black and red wires before plugging in a servo.

- Plug the servo that you can see in Figure 1.14 into servo port 12, and plug the other servo into servo port 13.

✓ **TIP**

The BOE Rev A does not have built-in servo ports. If you can not find the servo ports shown in Figure 1.15, go to Appendix D: Building Servo Ports on the Rev A Board of Education.

The Wheels

Figure 1.16 shows the Boe-Bot's wheel parts and mounting hardware.

Parts List:

(1) Partially assembled Boe-Bot (not shown)
(1) 1/16" Cotter pin
(2) O-ring tires
(1) 1" Polyethylene ball
(2) Plastic machined wheels
(2) Screws that attached the servo horns, which were set aside in the Removing the Servo Horns step.

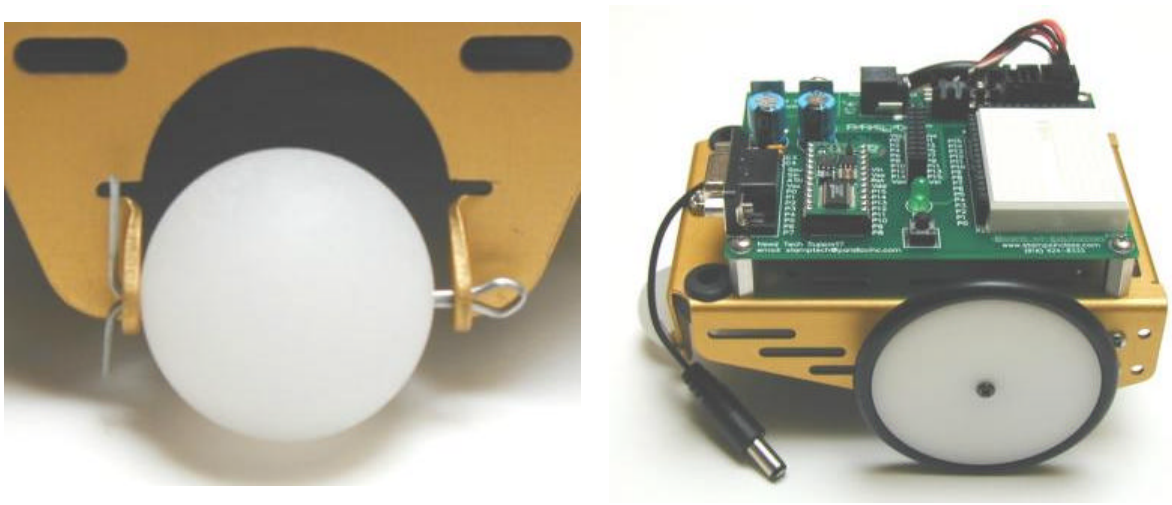
Figure 1.16: Wheel parts.

Assembly:

Figure 1.17 (a) shows the tail wheel attached to the Boe-Bot chassis with a cotter pin, and Figure 1.17 (b) shows one of the front wheels attached to a servo's output shaft.

- The plastic ball is used as the Boe-Bot's rear or tail wheel, and the cotter pin is its axle. Run the cotter pin through the holes in the tail of the Boe-Bot chassis so that it holds the one-inch plastic ball in place as shown in Figure 1.17 (a).

- Seat each o-ring tire in the groove on the outer edge of each plastic 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.



(a) (b)

Figure 1.17: (a), Tail wheel mounted on Boe-Bot chassis, and (b), front wheel mounted on servo output shaft.

Getting Connected

Figure 1.18 shows the parts you'll need to make your PC communicate with your BASIC Stamp 2.

Parts List:

(4) 1.5 V AA batteries
(1) Serial Cable
(1) Parallax CD

Figure 1.18: parts you'll need before your first program.

Chapter #1: Assembling and Testing Your Boe-Bot

Assembly:

Figure 1.19 shows the battery pack before and after the batteries are loaded.

- Load the batteries into the battery pack so that the polarity symbols on each battery match those printed on the inside of the battery pack.

Always use AA 1.5 V batteries.

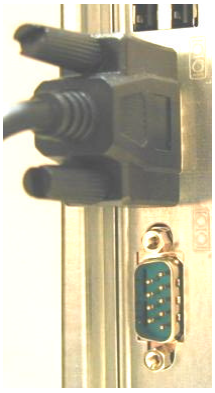
Do not use 1.2 V (typically rechargeable) batteries.

If you are considering using an AC adaptor that you can plug into the wall to power your Boe-Bot, see page 2 for details on which ones are compatible with your servos.

Figure 1.19: Battery pack without/with batteries.

Figure 1.20 shows (a), the serial cable connected to a COM port on the back of a PC, and (b) the serial cable and battery pack connected to the BOE.

- Plug the female end of the serial cable into one of your computer's unused serial ports.
- Plug the male end of the serial cable into the DB9 socket on the BOE.



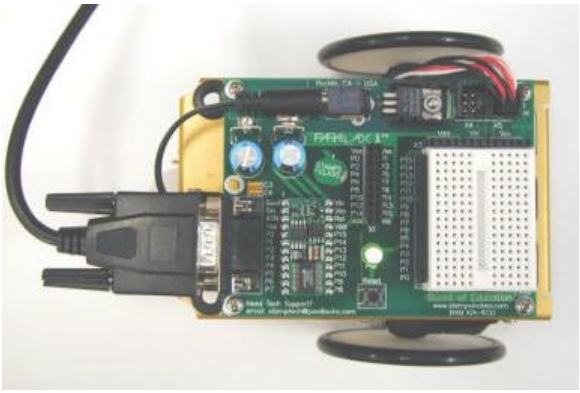


Figure 1.20: (a), Serial cable connected to com port, (b) BOE connected to serial cable and battery pack.

- Plug the battery pack back into the BOE while watching the green light on the BOE for problems. Unplug the battery pack immediately if you see any of the warning signs listed below.

Warning Signs:

If the green light doesn't come on, looks unusually dim, or flickers, disconnect the battery pack immediately and check your wiring. Any of these warning signs could indicate a wiring problem that could be dangerous to your servo and/or your BASIC Stamp.

- To extend the life of your batteries, unplug the battery pack's barrel plug from the BOE's barrel jack. This will disconnect power from the Board of Education and the servo motors. You will need to plug the power back in when you are ready to run your first PBASIC program in Activity #3.

Activity #3: Programming the Boe-Bot's BASIC Stamp 2 On-Board Computer

The Stamp Editor is the software you'll be using to program the Boe-Bot's BASIC Stamp 2 on-board computer. The Stamp Editor has a feature called the Debug Terminal. You can use the Debug Terminal to display messages received from the BASIC Stamp and also to send messages to the BASIC Stamp. The Debug Terminal will be one of your best and most used assistants for circuit testing and troubleshooting. Programming the BASIC Stamp to communicate with the Debug Terminal is very easy to do using the PBASIC programming language. We'll see shortly that the Debug command is very easy to use for displaying messages from the BASIC Stamp.

Software and First Program

This section covers the steps for:

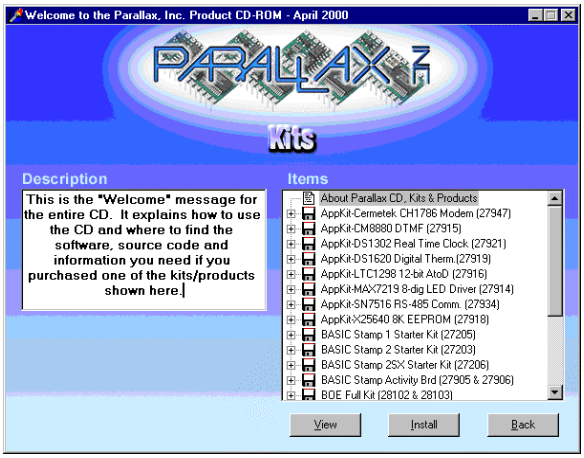
- Installing the Stamp Editor
- Using the Stamp Editor to establish PC – BASIC Stamp communication
- Running a sample PBASIC program that uses the **debug** command

Note: These instructions are for installing the Stamp Editor from the Parallax CD. A copy of the Parallax CD can be requested from stampsinclass@parallaxinc.com. You can also get the latest version of the Stamp Editor (It's free!) from the from Downloads page of www.parallaxinc.com.

Software

❑ If you have not already done so, load the Parallax CD into your computer's CDROM drive.

The Parallax CD has a browsing program called the Welcome application that runs automatically after the CD is placed in your computer's CDROM drive. Figure 1.21 (a) shows the browser as it appears the first time the CD is placed in the computer's drive. Figure 1.21 (b) shows the browser as it normally appears when you run the Welcome application.



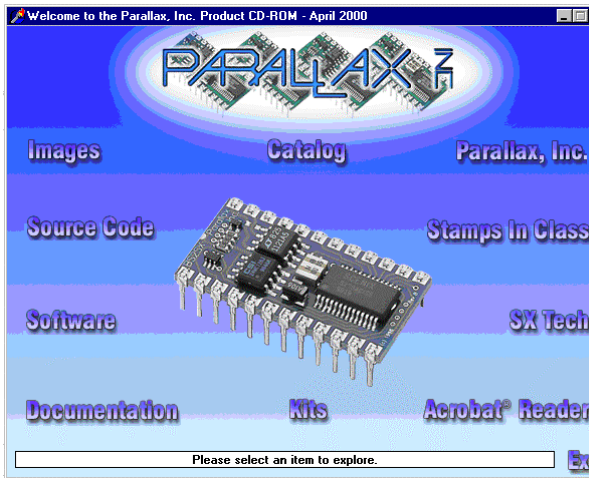


Figure 1.21: Welcome application (a) Kits page, and (b) Parallax page.

> ✓ **TIP**
>
> If the Welcome application did not run automatically, here's how to run it manually: Click the Start button on your Windows taskbar and select Run. When the Run window appears, enter the CDROM drive letter, followed by a colon, a backslash, and the name "Welcome.exe." For example, if the drive letter for your computer's CDROM drive is D, type in "D:\Welcome.exe." Click the OK button, and the Welcome application will run.

- ❑ If this is not your first time running the Welcome application, the Parallax page shown in Figure 1.21 (b) will display instead of the Kits page. Skip the next checklist instruction.
- ❑ If this is your first time running the Welcome application, a text document about the Parallax CD will automatically display. When you're finished reading the text document, minimize it or drag it out of the way so that you can see the Kits page. Click the "Back" button on the bottom right of the kits page to get to the Parallax page.
- ❑ Click the Software link.
- ❑ Click the + next to the BASIC Stamps folder, then click the + next to the Windows 95/98... folder, then click the diskette labeled Stamp 2/2e/2SX/2p (Stampw.exe).
- ❑ Click the Install button, and select "Yes" when the Confirm window asks you if you want to "Install selected files to C:\Parallax\Stamp?"
- ❑ If additional prompts appear, answer them as directed.

After installing the software, run it by following these steps:

- ❑ Click the Start button on your Windows taskbar and select Run.
- ❑ Enter "C:\Parallax\Stamp\Stampw.exe", and click OK.
- ❑ If this is your first time running the software, the Edit Port List similar to Figure 1.22 will appear. If you know the number of the COM port you're using and it doesn't appear in the list, enter the number in the Com # field, then click the Add button. If you know that a certain COM port listed in the Known Ports list is connected to a modem, click its entry in the list, then select Delete. Otherwise, just click OK.

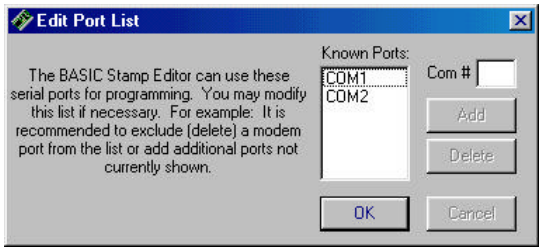


Figure 1.22: Edit Port List window.

✓ **TIP**	You can always modify this list later by going to the Preferences window. Just Click Edit, then select Preferences. Com port settings are under the Editor Operation tab.

The Stamp Editor window, similar to Figure 1.23 (a), will appear next. It will help to know the version number of the software you are using before you check to see if the Stamp Editor is communicating with the BASIC Stamp.

- Click the Help menu and select About…
- When the About window appears, make a note of the Version number.

Before attempting to run your first program, it's important to check and make sure the Stamp Editor can communicate with the BASIC Stamp.

- Plug the battery pack's barrel plug back into the barrel jack on the BOE. Verify the green light on the Board of Education lights up.
- Click the Run menu, and select Identify as shown in Figure 1.23 (b).

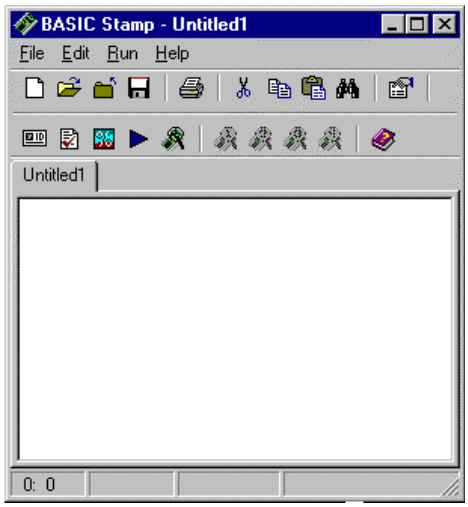


Figure 1.23: (a) Stamp Editor,

(b) Stamp Editor with Run | Identify selected.

Responses to Run -> Identify

When you click run and select Identify, the Stamp editor tries to find any BASIC Stamps connected to your PC's COM ports. If your Stamp Editor is Version 1.1 or lower, follow along in the left column below. If your Stamp Editor is Version 1.2 or higher, follow along in the right column.

Basic Stamp Editor, Version 1.1 or Lower

When everything is connected and working properly:

A window appears with the message that reads

- "Information: Found BS2-IC (firmware v1.0.)."

This message means the BASIC Stamp and PC are communicating. Continue to the next section, entitled "First Program."

Some of the other messages that might appear are:

- "Error: Basic Stamp II detected but not Responding…Check power supply."
- "Error: BASIC Stamp II not responding…Check serial cable connection. Check power supply."

If you get one of these error messages, follow the suggestions in the error message first. If that doesn't fix the problem, or if the error message you get doesn't offer any suggestions, go to Appendix B: PC to Stamp Communication Troubleshooting.

BASIC Stamp Editor Version 1.2 or Higher

An Identification window appears listing all the known COM ports and their status. One of the listed ports should read:

- "…BASIC Stamp 2 v1.0…"

This message means the BASIC Stamp and PC are communicating. Continue to the next section, entitled "First Program."

If the "Device Type" column is blank for every COM port listed, go to Appendix B: PC to Stamp Communication Trouble-Shooting.

First Program

Your first program will demonstrate the BASIC Stamp's ability to communicate with the outside world using the Debug Terminal. This handy terminal can be used for two-way communication between your PC and the BASIC Stamp. For now, we'll focus on programming the BASIC Stamp to send messages to the PC.

```
' Robotics! v1.5, Program Listing 1.1: Hello world!
' {$Stamp bs2}

debug "hello world"
```

- Type Program Listing 1.1 into the Stamp Editor as shown in Figure 1.24 (a).
- Click Run and select Run. Debug Terminal #1 should appear in a second window, as shown in Figure 1.24 (b).

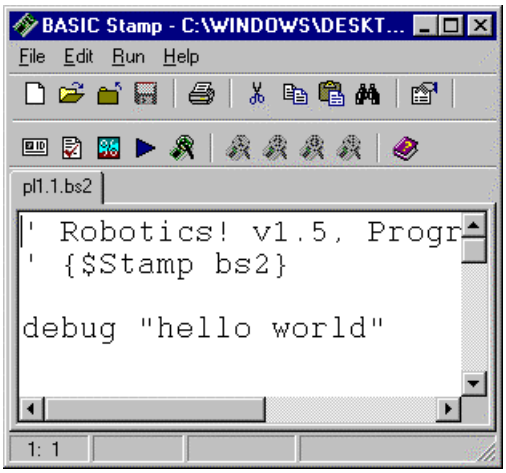


Figure 1.24: (a) Stamp Editor **(b) Debug Terminal.**

How "Hello world!" Works

The apostrophe character was used in two ways in this program. The first line used the apostrophe to include a comment in the program. A comment is text that the programmer adds to help document the program and make it more understandable. So the line of code that reads:

```
' Robotics! v1.5, Program Listing 1.1: Hello world!
```

is a comment. The special thing about the comment is that you can type just about anything to the right of the apostrophe. In this case, we typed the name of the book this program is in, the number of the program in the book and a very brief description.

There is one special directive that has meaning to the Stamp Editor even though it gets placed in a commented line in your program. It's called the Stamp Directive, and it tells the Stamp Editor what kind of BASIC Stamp you are writing programs for. The Stamp Directive is in the second line of the program, and it reads:

```
'{$Stamp bs2}
```

This special directive tells the Stamp Editor that the program should be downloaded to a BASIC Stamp 2. There are several different Stamp Directives that are used with the different BASIC Stamps manufactured by Parallax, Inc. The Boe-Bot always comes with the BASIC Stamp 2, so you will see that every Program Listing in this workbook uses the Stamp Directive for the Basic Stamp 2.

The third line in the program is the only PBASIC command in the program, the **debug** command. This command caused the Debug Terminal window to appear and display the "hello world" message. When a program containing a **debug** command is run, the Stamp Editor opens a Debug Terminal. When the BASIC Stamp executes the command:

```
debug "hello world"
```

it sends the "hello world" message to your PC by way of the serial cable. The "hello world" message is a text string, which is one of several types of output data the BASIC Stamp can be programmed to send using the **debug** command.

Your Turn

The best way to get a better feel for what you can do with the **debug** command is to try the examples in the BASIC Stamp Manual. When you turn to the **debug** command section in the BASIC Stamp Manual, you'll see lots of program examples. The next few paragraphs are excerpts from the *BASIC Stamp Manual, v2.0c* with a couple of **debug** examples you can try. Note that there are Stamp™ icons next to the section heading that reads "BASIC Stamp 2, 2e, 2sx, and 2p Formatting". Each of these icons has a number in it, and the ones with the "2" inside mean the section/example applies to the BASIC Stamp 2. Make sure that the examples you try are from sections that apply to the BASIC Stamp 2; otherwise, you might get some confusing error messages.

- Try the **debug** command examples in this excerpt. Make sure to include the Stamp Directive somewhere in the Stamp Editor along with the code segment before you click the Run menu and select Run.

BASIC Stamp 2, 2e, 2sx and 2p Formatting

DISPLAYING ASCII CHARACTERS.

On the all BASIC Stamps except the BS1, the DEBUG command, by default, displays everything as ASCII characters. What if you want to display a number? You might think the following example would do this:

```
x     VAR    BYTE
x = 65
DEBUG  x                          ' Try to show decimal value of x.
```

DISPLAYING DECIMAL NUMBERS.

Since we set *X* equal to 65 (in line 2), you might expect the DEBUG line to display "65" on the screen. Instead of "65", however, you'll see the letter "A" if you run this example. The problem is that we never told the BASIC Stamp how to output *X*, and it defaults to ASCII (the ASCII character at position 65 is "A"). Instead, we need to tell it to display the "decimal form" of the number in *X*. We can do this by using the decimal formatter (DEC) before the variable. The example below will display "65" on the screen.

```
x     VAR    BYTE
x = 65
DEBUG  DEC  x                    ' Show decimal value of x.
```

- Look up the **debug** command in the BASIC Stamp Manual's table of contents, and continue through the **debug** command examples in the BASIC Stamp Manual's **debug** command section. Remember to only try examples in sections that discuss the BASIC Stamp 2. Remember also to add the Stamp Directive: **' {$Stamp bs2}**.

Activity #4: Testing the Servos Individually

In this activity, you will program the BASIC Stamp to control the rotation of each of the Parallax pre-modified servos on the Boe-Bot.

How Servos Work

Normal (un-modified) hobby servos are very popular for controlling the steering systems in radio-controlled cars, boats, and planes. These servos are designed to control the position of something such as a steering flap on a radio-controlled airplane. Their range of motion is typically 90° or 180°, and they are great for applications where inexpensive, accurate high-torque positioning motion is required. The position of these servos is controlled by an electronic signal called a pulse train, which you'll get some first hand experience with shortly. An un-modified hobby servo has built-in mechanical stoppers to prevent it from turning beyond its 90° or 180° range of motion. It also has internal mechanical linkages for position feedback so that the electronic circuit that controls the DC motor inside the servo knows where to turn to in response to a pulse train.

FYI	For experiments with unmodified servos, try What's A Microcontroller? Experiments 3 and 4, available for free download from the www.stampsinclass.com -> Downloads -> Educational Curriculum page. You will need an unmodified hobby servo and a few additional parts for these experiments.

A Parallax pre-modified servo does not have the position feedback and mechanical stoppers you find in normal hobby servos. You can send the same electronic signals (a pulse train) to a Parallax pre-modified servo as you would normally send to a hobby servo. In a hobby servo, a given pulse train makes it turn to a certain position and stay there. The same pulse train causes a Parallax pre-modified servo to turn continuously. The pulse train also sets the pre-modified servo's speed and direction. So, instead of controlling airplane flaps, the Parallax pre-modified servos are used as BASIC Stamp controlled motors that make the Boe-Bot's wheels turn.

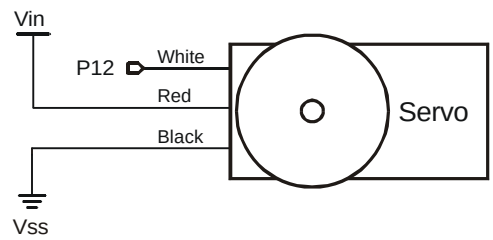


Figure 1.25: Servo connection schematic.

Figure 1.25 shows the circuit that is established when a servo is plugged into the servo port labeled 12 on the BOE Rev B's top right corner. The red and black wires connect to the servo's power source, and the white (or sometimes yellow) wire is connected to a signal source. When a servo is plugged into servo port 12, the servo's signal source is BASIC Stamp I/O pin P12.

About Time Measurements and Voltage levels

Throughout this student workbook, amounts of time will be referred to in units of seconds (s), milliseconds (ms), and microseconds (µs). Seconds are abbreviated with the lower-case letter s. So, one second is written as 1 s. Milliseconds are abbreviated as ms, and it means one one-thousandth of a second. One microsecond is one one-millionth of a second. The Milliseconds and Microseconds box to the right shows these equalities in terms of both fractions and scientific notation.

Milliseconds and Microseconds

$$1\,\text{ms} = \frac{1}{1000}\,\text{s} = 1\times10^{-3}\,s$$

$$1\,ms = \frac{1}{1{,}000{,}000}\,\text{s} = 1\times10^{-6}\,s$$

Voltages and BOE Labels

$$\text{Vss} = 0\text{ V (ground)}$$
$$\text{Vdd} = 5\text{ V (regulated)}$$
$$\text{Vin} = 6\text{ V (unregulated)}$$

A voltage level is measured in volts, which is abbreviated with an upper case V. The BOE has sockets labeled Vss, Vdd, and Vin. Vss is called the system ground or reference voltage. When the battery pack is plugged in, Vss is connected to its negative terminal. As far as the BOE, BASIC Stamp and serial connections to the computer are concerned, Vss is always 0 V. Vin is unregulated 6 V, and it's connected to the positive terminal of the battery pack. Vdd is regulated to 5 V by the BOE's onboard voltage regulator, and it will be used with Vss to supply power to circuits built on the BOE's breadboard.

Only use the Vdd sockets above the BOE's breadboard for the Activities in this workbook. Do not use the Vdd on the 20-pin app-mod header.

The control signal the BASIC Stamp sends to the servo's control line is called a "pulse train," and an example of one is shown in Figure 1.26. The BASIC Stamp can be programmed to produce this waveform using any of its I/O pins. In this activity, we'll start with I/O pin P12, which is already connected to servo port 12 by a metal trace built into the Board of Education. First, the BASIC Stamp sets the voltage at P12 to 0 V (low) for 20 ms. Then, it sets the voltage at P12 to 5 V (high) for 1.0 ms. Then, it starts over with a low output for another 20 ms, and a high output for another 1.0 ms, and so on.

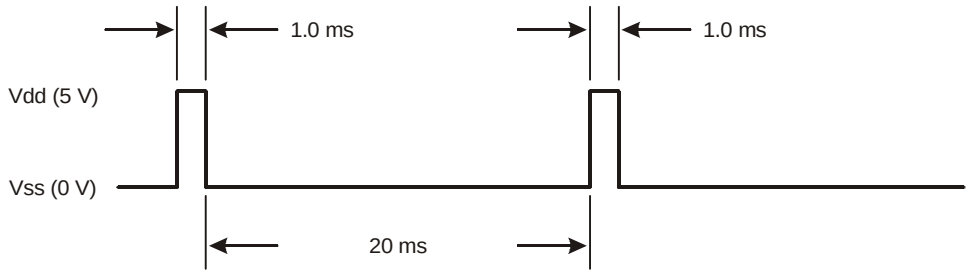


Figure 1.26: Pulse train.

This pulse train has a 1.0 ms high time and a 20 ms low time. The high time is the main ingredient for controlling a servo's motion, and it is most commonly referred to as the pulse width. In this example, we are working with 1 ms wide pulses. Since these pulses go from low to high (0 V to 5 V) for a certain amount of time, they are called positive pulses. Negative pulses would involve a resting state that's high with pulses that drop low. Pulse trains have some other technical descriptions such as duty and duty cycle. These are described in BASIC Analog and Digital, Experiment #6.

Remember	Pulse width is what controls the servo's motion. The low time between pulses can range between 10 and 40 ms without adversely affecting the servo's performance.

A pre-modified servo can be pulsed to make its output shaft turn continuously. The pulse widths for pre-modified servos range between 1.0 and 2.0 ms for full speed clockwise and counterclockwise respectively. If you give a pre-modified servo 1.25 ms pulses, it will turn clockwise at roughly half of full speed. If you give a pre-modified servo 1.90 ms pulses, the servo will turn at almost full speed counterclockwise. The "center

pulse width" is 1.5 ms, and that makes the servo stay still. If the servo turns very slowly in response to 1.5 ms pulses, you will learn how to adjust the servo to stay still using a PBASIC program in Activity #6.

Figure 1.27 shows the Boe-Bot's front, back, left and right. Use this diagram as your guide when you see instructions about making the Boe-Bot move forward/backward, examining the right or left wheels, etc.

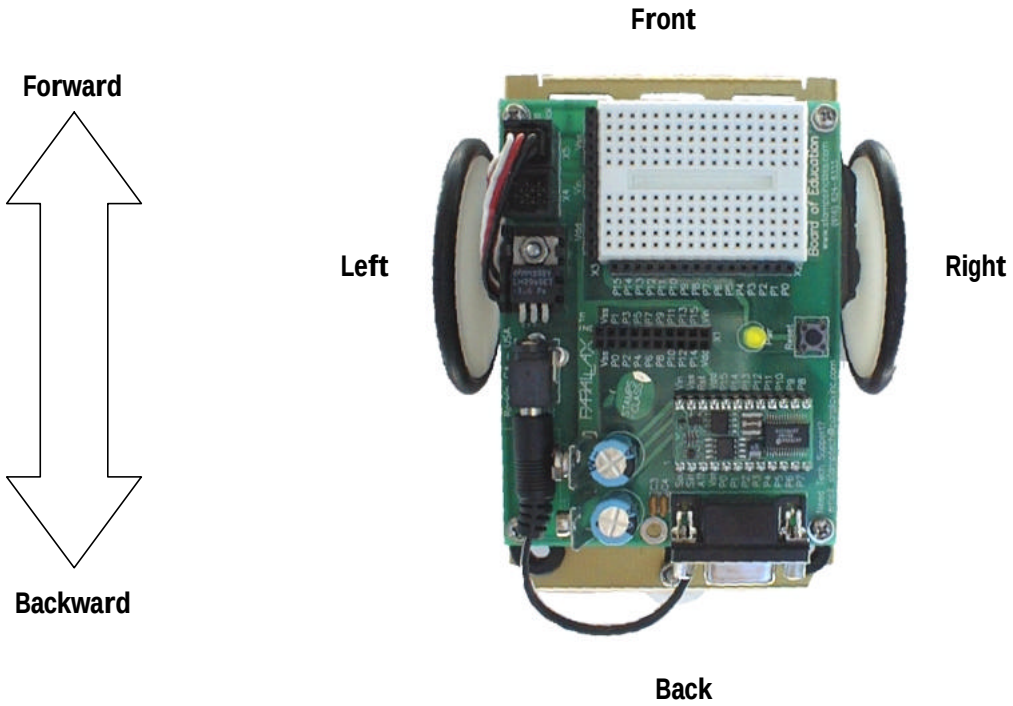


Figure 1.27: Boe-Bot from the driver's seat.

Let's start by programming the Boe-Bot's right wheel to turn full speed ahead. For the right side of the Boe-Bot, this means the wheel has to turn clockwise, which means it needs to receive 1 ms pulses every 20 ms or so.

- You may want to set the Boe-Bot on something to keep it's wheels from touching the ground during these tests. Otherwise, you will see the Boe-Bot spin around in circles since only one wheel is turning.

- Enter Program Listing 1.2 into the Stamp Editor.

```
' Robotics! v1.5, Program Listing 1.2: Right Wheel Full Speed Ahead.
' {$Stamp bs2}                    ' Stamp Directive.

low 12                            ' Set P12 to output-low.

loop:                             ' Target label for last command - “goto loop”.

  pulsout 12, 500                 ' Send 1.0 ms pulses to P12
  pause 20                        ' every 20 ms.

goto loop                         ' Send program to “loop:” label.
```

- Save the program using a descriptive name, such as “Boe-Bot 1.2”. You can do this by clicking the File Manu and selecting Save (or Save As... if you are renaming the file). Then enter the “Boe-Bot 1.2” into the File name: field, and make sure that the Save as type: field is set to “BASIC Stamp 2 files (*.bs2)”.

- Run the program by clicking Run and selecting Run.

- Verify that, as you're looking at the wheel from the Boe-Bot's right side that it is turning clockwise fairly rapidly (about 37 RPM).

How the Right Wheel Full Speed Ahead Program Works

- Look up each of the following new commands in Appendix C: PBASIC Quick Reference or in the BASIC Stamp Manual before continuing: **`low, pulsout, pause, goto.`**

As with the previous program example, the first line of the program is a descriptive comment and the Stamp Directive is on the second line. In this program there are also comments (that begin with apostrophes) to the right of each PBASIC command. With the exception of the Stamp Directive, when entering the commands into the Stamp Editor, you don't need to include the comments. Aside from the Stamp Directive, comments and other ways of documenting your programs become important later on if you start writing code that's more complex or that somebody else needs to work with. For the time being, just enter the commands (and the Stamp Directive) into the Stamp Editor unless otherwise directed by your instructor.

The command **`low 12`** does two things. It sets BASIC Stamp I/O pin P12 to output, then it sets its output value low. As an output, P12 can send voltage signals, as opposed to being set to **`input`**, which means P12 would listen for signals instead. Setting the output value low means that the voltage P12 sends is the same as Vss, 0 volts. If the command **`high 12`** were used instead, P12 would send a high signal, which would be Vdd, 5 volts.

When a word that's not a PBASIC command is followed by a colon, it's a called a "label". As you get more and more familiar with PBASIC, you'll start to recognize which words are commands and which words are labels automatically. The **`loop:`** label works together with the command **`goto loop`** at the end of the program. The **`loop:`** label is a place holder, and whenever the program gets to the command **`goto loop`**, it causes the program to start executing commands at the **`loop:`** label again. The result is that the **`pulsout`** and **`pause`** commands get executed over and over, which causes a continuous pulse train to be sent to the servo.

The **`loop:...goto loop`** program structure is called an infinite loop. An infinite loop means that part of the code in the program is executed over and over again with no code that allows it to stop repeating the same set of instructions. Often with normal computer programs, this is a problem. However, infinite loops are commonly used in microcontroller programming. In fact, most microcontroller programs, including the ones in this text, are written within the framework of an infinite loop. With the BASIC Stamp, you can always end an infinite loop by disconnecting the power or using the Stamp Editor to run a different program.

The command **`pulsout 12, 500`** sends a 1.0 ms pulse to P12. The **`pulsout`** command has two arguments, ***`pin`*** and ***`period`***. The ***`pin`*** argument refers to the I/O pin the BASIC Stamp sends the voltage pulse to, and ***`period`*** refers to how long the voltage pulse lasts. The I/O pin number makes sense; the number 12 refers to I/O pin P12. What about the ***`period`*** argument of **`500`**? How does that correspond to a 1.0 ms pulse? The answer is that the ***`period`*** argument for the **`pulsout`** command has to be specified in 2 µs increments. So, if you want the pulse to last for 1.0 ms, you have to use a number that gives you 1.0 ms when it's multiplied by 2 µs increments. Here is proof that a **`pulsout`** ***`period`*** of **`500`** fits the bill.

$$
\begin{aligned}
500 \times 2\ \mu s &= 500 \times (2 \times 10^{-6})\, s \\
&= 1000 \times 10^{-6}\, s \\
&= 10 \times 10^{-3}\, s \\
&= 10\ ms
\end{aligned}
$$

The command **`pause 20`** is much more obvious. That's because the ***`period`*** argument for the **`pause`** command is specified in 1 ms increments. So, if you want a 20 ms pause, **`pause 20`** does the job.

Your Turn

Program Listing 1.3 is really Program Listing 1.2 with one small change. Instead of **`pulsout 12, 500`**, Program Listing 1.3 uses the command **`pulsout 12, 1000`**. This should make the right wheel turn full speed counterclockwise.

- ❑ Run Program Listing 1.3 as shown.

```
' Robotics! v1.5, Program Listing 1.3: Right Wheel Full Speed Reverse.
' {$Stamp bs2}                    ' Stamp Directive.

low 12                            ' Set P12 to output-low.

loop:                             ' Label for "goto loop to return to"

  pulsout 12, 1000                ' Send 2.0 ms pulses to P12
  pause 20                        ' every 20 ms.

goto loop                         ' Send program to "loop: " label.
```

- ❑ To make the right wheel stay still, modify the **`pulsout`** command so that it reads **`pulsout 12, 750`**, and run the modified program.

Program Listing 1.4 is Program Listing 1.3 with two small changes:

- **`low 12`** was changed to **`low 13`**
- **`pulsout 12, 1000`** was changed to **`pulsout 13, 1000`**

- ❑ Run Program Listing 1.4 as shown to make the left wheel turn full speed counterclockwise.

```
' Robotics! v1.5, Program Listing 1.4: Left Wheel Full Speed Ahead.
' {$Stamp bs2}                    ' Stamp Directive.

low 13                            ' Set P12 to output-low.

loop:                             ' Label for "goto loop to return to"

  pulsout 13, 1000                ' Send 2.0 ms pulses to P13
  pause 20                        ' every 20 ms.

goto loop                         ' Send program to "loop: " label.
```

- To make the left wheel turn full speed clockwise, modify the command **`pulsout 13, 1000`** so that it reads **`pulsout 13, 500`**, and run the modified program.

- Now, change the ***`period`*** argument of the **`pulsout`** command from **`500`** to **`750`**, and the wheel should stay still.

Activity #5: Running Both Servos

When you assembled the Boe-Bot in Activity #2, you plugged the servo on the right side of the Boe-Bot into P12 and the servo on the left side of the Boe-Bot into P13. Figure 1.28 shows a schematic of the circuit you created by doing this. The servo on the Boe-Bot's right side is connected to I/O line P12 and the servo on the Boe-Bot's left is connected to P13. Each servo is also connected to Vin (the battery pack's positive terminal) and Vss (the battery pack's negative terminal).

The easy part about making the Boe-Bot roll forward is that you include two **`pulsout`** commands, one for each servo. The difficult part can be figuring out what the ***`period`*** arguments should be.

Take a look at the right side of the Boe-Bot. To make this wheel turn forward, the servo has to turn clockwise. This means a ***`period`*** argument less than center. A ***`period`*** of 500 will work well for full speed ahead. Now look at the left side of the Boe-Bot. To make this wheel turn forward, the servo has to turn counterclockwise. Now instead of 500, a ***`period`*** of 1000 is needed.

- Enter and run Program Listing 1.5 and observe the results.

If the Boe-Bot rolled backward instead of forward, the servo lines were swapped. It means that the servo plugged into servo port 12 should be plugged into servo port 13 and visa-versa.

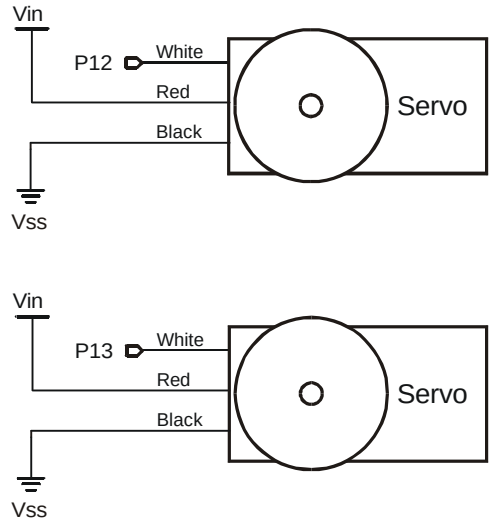


Figure 1.28: Servo connection schematic.

```
' Robotics! v1.5, Program Listing 1.5: Full Speed Ahead – both servos.
' {$Stamp bs2}                    ' Stamp Directive.

low 12                            ' Set P12 to output-low.
low 13                            ' Set P13 to output-low.

loop:                             ' Label for "goto loop to return to"

  pulsout 12, 500                 ' Send 1.0 ms pulses to P12
  pulsout 13, 1000                ' Send 2.0 ms pulses to P13
  pause 20                        ' every 20 ms.

goto loop                         ' Send program to "loop: " label.
```

How Program Listing 1.6 Pulses Both Servos

Program Listing 1.6 pulses the left servo, then the right servo, then pauses. It does this in an infinite loop, so the Boe-Bot just keeps going. Among other things, in Chapter 2, we'll work on setting the distance traveled.

Your Turn

- After you make each change listed below, make sure to run the modified version of the program and observe what the Boe-Bot does differently.
- Swap the **`pulsout`** ***`period`*** arguments to make the Boe-Bot to roll backward. In other words, instead of using the commands **`pulsout, 12, 500`** and **`pulsout 13, 1000`**, use the commands **`pulsout 12, 1000`**, and **`pulsout 13, 500`**. This should make the Boe-Bot travel backward instead of forward.
- Try setting both **`pulsout`** ***`period`*** arguments to the center value of 750 to make the Boe-Bot stay still.
- Try setting both **`pulsout`** ***`period`*** arguments to 500 and run the modified program. It will make the Boe-Bot rotate counterclockwise in place.
- Try setting both **`pulsout`** ***`period`*** arguments to 1000 and run the modified program. It will make the Boe-Bot rotate clockwise in place.

Activity #6: Tuning the Servos – Calibration in Software

Chances are that you noticed your Boe-Bot didn't go perfectly straight forward when you ran Program Listing 1.5. For that matter, it probably didn't go perfectly straight backward in response to the modifications you made to Program Listing 1.5 in the Your Turn section of Activity #5. You can adjust a **`pulsout`** command's ***`period`*** argument to straighten out the Boe-Bot's travel. This practice is called "calibration in software".

If the Boe-Bot veers to the right when it is programmed to go straight forward, either the left wheel needs to slow down, or the right wheel needs to speed up. Since the servos are pretty close to top speed as it is, slowing the left wheel down will work better. You can do this by making the pulse period to the left servo, which is connected to P13, smaller. For example, instead of using the command **`pulsout 13, 1000`**, you might try **`pulsout 13, 940`**. Keep in mind that the adjustment is different if you need to slow the right wheel's rotation. In that case, use a **`pulsout`** ***`period`*** argument larger than 500, such as 560 for starters. By trying different values, you can home in on the value that will get your Boe-Bot wheels turning forward at the same speed, then your Boe-Bot will move forward in a straight line.

- ❑ The paragraph that comes just before this checkbox instruction describes how to modify Program Listing 1.5 to make the Boe-Bot travel forward in a straight line. Test your Boe-Bot and try these modifications. Keep in mind that you may have to try many different **`pulsout`** ***`period`*** values to fine tune the Boe-Bot's motion.

You can also make the same corrections to get the Boe-Bot travel in a straight line backward. You will first need to modify Program Listing 1.5 to make the Boe-Bot travel full speed backward. Then, test to figure out which wheel needs to slow down. Since the **`pulsout`** ***`period`*** values have to be swapped to make the Boe-Bot travel full speed reverse, you will also need to adjust the **`pulsout`** ***`period`*** argument differently. For a given wheel, slowing it down involves taking the full speed **`pulsout`** ***`period`*** argument, and adjusting it so that it is slightly closer to the center **`pulsout`** ***`period`*** of 750.

- ❑ Make the necessary changes in Program Listing 1.5 so that it makes the Boe-Bot go full speed backward.

- ❑ Make the additional adjustments to one of the two **`pulsout`** commands to slow down the wheel that's turning too fast to cause the Boe-Bot to travel backwards in a straight line.

IMPORTANT: You can re-use the **`pulsout`** ***`period`*** values you just determined in later activities. There will be many example programs in this workbook where the values of 1000 and 500 are used for full speed ahead and full speed reverse. You can substitute your "calibration in software" values for better results when you try the rest of the example program listings in this workbook.

- ❑ Make notes of the fine-tuned **pulsout *period*** values you came up with for full speed straight forward and full speed straight backward.

You can skip the rest of this activity if your servos remained still when you modified Program Listings 1.3 and 1.4 to send pulse widths of 1.5 ms. This calibration involves testing each servo at pulse periods around the predicted center **pulsout *period*** value of 750 (1.5 ms). What you'll be looking for is the best ***period*** value to really make the servo stay still.

- ❑ First, run Program Listing 1.6. If the servos don't move, they are calibrated. If the servos rotate slowly, follow the instructions below. If the servos turn rapidly, check your program for clerical errors.

```
' Robotics! v1.5, Program Listing 1.6: Stay still (centered) – both servos.
' {$Stamp bs2}                    ' Stamp Directive.

low 12                            ' Set P12 to output-low.
low 13                            ' Set P13 to output-low.

loop:                             ' Label for "goto loop to return to"

  pulsout 12, 750                 ' Send 1.5 ms pulses to P12
  pulsout 13, 750                 ' Send 1.5 ms pulses to P13
  pause 20                        ' every 20 ms.

goto loop                         ' Send program to "loop: " label.
```

By viewing the slowly turning Boe-Bot wheel from the side, you can decide whether to search for the true center **pulsout *period*** using values above or below 750.

- If the wheel is turning clockwise, that means that its true center **pulsout *period*** is somewhere above 750.
- If the wheel is turning counterclockwise, it means the true center **pulsout *period*** is somewhere below 750.

The test for finding the correct center pulse width for each servo involves finding the range of values that make the servo stay still. If the servo is turning slowly clockwise, try **pulsout *period*** values of 751, 752, 753,... To figure out where the true center is, you will need to keep on increasing the **pulsout *period*** argument. Make a note of the first value that causes the servo to stay still. Then, keep increasing the **pulsout *period*** until the servo starts turning the opposite direction. Make a note of that value as well. Keep in mind that this process works almost the same for a servo that is turning counterclockwise, simply look for the true center pulse width somewhere below 750. Your search will begin at 749, then continue downwards until the servo stops turning, then starts turning the opposite direction.

After finding the upper and lower **pulsout *period*** values that make your servo stay still, you can take the average of those two values to calculate the true center **pulsout *period*** argument. This calculated value is the one you should use in place of 750 in Program Listings.

- For each servo that does not stay still when you run Program Listing 1.6, perform the procedure just described for finding the **pulsout *period*** value that makes the servo stop turning, and the value that makes the servo start turning again in the opposite direction.

The true value of the ***period*** argument to center the servo is somewhere between when the servo stops and when the servo starts again. For example, if the servo stops turning at 753 and starts turning again at 755, it's easy. Just take 754 as the center ***period*** for that servo.

Here's an example that's a little more difficult. What if the servo doesn't stop turning until it gets a ***period*** argument of 752? Then, what if it doesn't start turning again until it gets a ***period*** argument of 757? The true ***period*** to center this modified servo is would be somewhere between 752 and 757, but where? By taking the average of the two numbers, the ***period*** would be:

$$\text{average pulsout period} = \frac{752 + 757}{2} = 754.5$$

The **pulsout** command's ***period*** argument can only be specified by an integer value between 0 and 65535. Some examples of valid ***period*** arguments are: 0, 1, 2,...,754, 755,.... So, 754.5 can not be used as a valid ***period*** argument. The average **pulsout *period*** needs to be rounded, but which way? Although the standard practice is to round up if the decimal value is 0.5 or above, the servo might not really behave according to that practice. You could experimentally determine which way to round by trying periods of **758** and **751** in your program. Continuing the example, if 751 makes the servo to turn faster, round upward, and take 755 as the center ***period***. On the other hand, if 758 makes the servo turn faster, round downward, and take 754 as the center ***period***.

- Use the calculations just described to figure the true center **pulsout *period*** argument for each servo that did not stay still when you ran Program Listing 1.6.

FYI	If all attempts to calibrate your servo in software fail, you can download instructions for how to disassemble and adjust it from the www.stampsinclass.com -> Robotics! page.

- Add the **pulsout *period*** arguments for the center ***period*** values to your notes on the ***period*** values for the straight forward and backward values you've already gathered.

Summary and Applications

Congratulations on the construction and operation of your Boe-Bot! Through following the procedures in this chapter, you may have had your first taste of testing and troubleshooting at the system and subsystem levels. Lots of other essential topics were covered that will get used and re-used throughout this text. For example, the Debug Terminal will be your best and most used tool for testing and troubleshooting each circuit as well as many upcoming programs.

The PBASIC programming language was introduced along with some example programs to get you started with the Debug Terminal and with the Boe-Bot. Comments and labels were introduced, as were the commands **`low, pulsout, pause`**, and **`goto`**. Software calibration also was introduced.

Real World Example

From the space shuttle all the way down to the Boe-Bot, isolating and testing subsystems during each phase of development is critical to make sure the whole thing runs when it's put together. More importantly, isolating and testing each subsystem minimizes the time spent on, and difficulty level of, troubleshooting. At the beginning of the chapter, the problems associated with not iteratively developing and testing were discussed. Imagine if nobody tested the Space Shuttle's subsystems before putting it together. It would take hundreds of years for NASA to get all their problems sorted out!

Whether it's robotics competitions, product development, or space programs, subsystem and system level development and testing is the way to avoid unnecessary delays when working from the beginning to the end of a project. Especially in product development, groups of engineers develop systems and subsystems. Often, it's not until late in the design cycle that the system level testing and system integration occurs. Sometimes, all a design team knows are the input and output (I/O) requirements of their particular module in the project. Regardless, engineering design teams still have to iteratively develop, simulate (which we did not do here), and test the subsystems within the project module they are working on.

Software calibration also was introduced. This is currently a hot topic because many appliance makers are working on incorporating microcontrollers that can communicate across the Internet into their products. Remote diagnostic programs can then enhance the ability of the microcontrolled devices to self calibrate. Also, technicians can perform software calibration, diagnosis, and in some cases repair, all remotely. Imagine your TV picture going bad a few years from now. Getting it fixed might involve the pressing of a few buttons on your TV remote. Then the microcontroller in your TV will log onto the Internet and report the problem to the company that makes the TV. Before sending out a TV repair person, your TV's problem will be diagnosed, perhaps by a computer program. It might even be fixed by the program sending special control instructions back to the microcontroller. Otherwise the TV repair person will show up with the right parts to fix it.

Boe-Bot Application

One item you'll investigate in the Questions and Projects section is what happens when the wiring of the servos gets changed. How does this get handled? It involves more changes than you might think. For example, if you were to unplug a servo from servo port 12 and then plug it into servo port 15, you can't just change the drawing that shows what port to plug it into. The schematic, which is the preferred method of communicating wiring information, has to be changed, but so do all the program listings. At some point you might want to add more servos to function as grippers. Although a gripper design is not included in this text, Questions and Projects has exercises that will prepare you for connecting servos to different ports.

So far, the Boe-Bot can be programmed to roll forward or backward or to rotate in place. During some of your testing, small variations in servo performance were discovered and corrected in software. The advantage of calibration in software is that, instead of mechanically adjusting the Boe-Bot, the PBASIC program is modified to make the correction. This software calibration was all done at only two or three speeds: full speed ahead, full speed reverse, and full stop. In the projects section you'll get a chance to further research and generalize the software calibration for a variety of speeds.

Questions and Projects

Questions

1. Explain the two things the **`debug`** command does.

2. What are the different types of output data that can be used with the **`debug`** command? Hint: Use the BASIC Stamp Manual to answer this question.

3. Assuming the circuit inside the servo that controls the DC motor updates what it's doing every 20 ms, how many times per second does the DC motor get updated? Hint: This is a division problem.

4. Discuss how you would modify Figure 1.25 and Program Listing 1.2 if you wanted to test each of your servos using I/O line P15. How would it change Figure 1.15 (b)?

5. Assume in Activity #5 that you want to use I/O lines P14 and P15 to drive both servos. Make sketches of your recommended modifications to the relevant diagrams, pictures, and code samples in this activity to accommodate this change.

6. What would you do if you wanted to drive the servos using I/O lines P10 and P11. Repeat the activities in Question 5. Hint: One of the Appendix sections in the back of the book will be useful.

Exercises

1. What would happen if you used **`pause 30`** instead of **`pause 20`** in the Chapter #1 program listings? How would this effect the servos' operation? Draw a diagram similar to Figure 1.26 based on a pulse train using **`pause 30`** and **`pulsout 12, 1000`**.

2. The How the Right Wheel Full Speed Ahead Program Works section in Activity #4 has a math example that shows how the **`pulsout`** ***`period`*** argument 100 equates to 1.0 ms pulses. Repeat this exercise for **`pulsout 12, 750`** and **`pulsout 12, 1000`**.

3. What is the necessary **`pulsout`** ***`period`*** argument to make a 1.626 ms pulse?

4. What are the maximum and minimum pulse widths that can be generated using the **`pulsout`** command?

5. Challenge: Since the low time in the pulse train is not overly important, the **`pause`** ***`period`*** was not adjusted when a second **`pulsout`** command was added. However, each **`pulsout`** command causes an additional delay on top of the 20 ms pause. How would you adjust the **`pause`** ***`period`*** in Program Listing 1.5 to ensure that the pulse to a given servo begins every 20 ms?

Projects

1. In Question 5, you modified the relevant figures and programs in Activity #5 so that you could use I/O lines P14 and P15 to drive the servos instead of I/O lines P12 and P13. Use your Boe-Bot to test these modifications. Make sure to save your programs under different file names, because, in the next chapter, we'll start over again with I/O lines 12 and 13.

   Record the Boe-Bot's behavior on the first run of your test. Was this the expected behavior? If not, account for it, then record the additional changes you had to make to get your Boe-Bot working as intended. There may be multiple iterations of troubleshooting here; make sure to record each step taken to get the Boe-Bot to function as intended.

2. Program the Boe-Bot to move in several different patterns. Try the following:

   (a) Identify a pair of **`pulsout`** values that make the Boe-Bot move slowly straight forward. Shoot for wheel speeds of 4 revolutions per minute (RPM). Some trial and error will be necessary to find the **`pulsout`** ***`period`*** value to make each servo turn at this rate.
   (b) Identify a pair of **`pulsout`** ***`period`*** values that make the Boe-Bot move very slowly straight backward at the same speed. How do these values compare (or not compare) to those identified in 2 (a.)?

3. Make a graph of wheel speed as a function of pulse width for each servo. Use several pulse widths between 0.8 and 2.2 ms (**`pulsout`** values between 400 and 1100). Either count how many revolutions the wheel completes in a specified time (20 seconds or a minute), or see how much time it takes to complete 10 revolutions.

   Your graph might look similar to Figure 1.29. This graph was generated by a Microsoft Excel spreadsheet using eight data points and the "Best Fit" option. Collect data points and make your own graphs, one for each servo. In general, when you plot more data points, you can expect your graph to be more reliable. However, there are lots of techniques for reducing the number of measurements. One example would be to take a few measurements to find the curved areas of the graph, then focus

on taking many measurement in those areas while taking only a few measurements in the areas that are linear.

Note: Expect your graphs to look different from Figure 1.29 because it's for a different kind of servo from the one in your kit.

Use your graphs to predict the pulse widths required to make your Boe-Bot go straight forward or straight backward with less trial and error. Try this at a variety of speeds. The rotational speed of one servo will correspond with a certain **`pulsout`** value. Remember that the values you select will come from opposite sides of the 750 center **`pulsout`** ***`period`***.

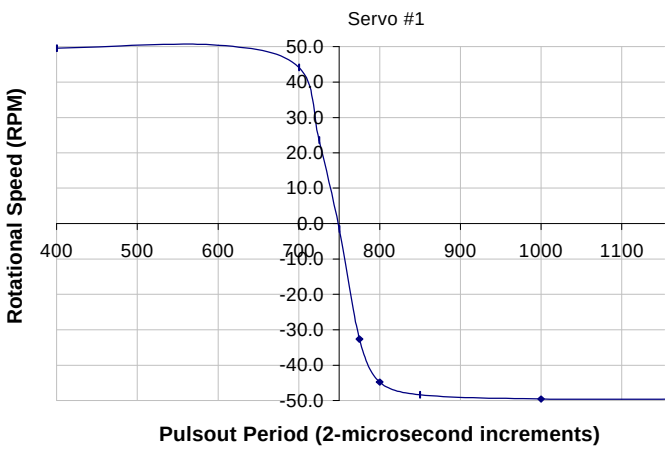


Figure 1.29: `pulsout period` vs. rotational speed for servo.

Test the accuracy of the graphically determined predictions by programming them into your Boe-Bot and testing to see how straight it goes. By focusing on one servo for your corrections, you can calculate the percent error of your initial guess. So that error from the other servo is not a factor, fine tune it using trial and error so that it rotates at the exact speed predicted by your graphs. Then, you can figure the percent error on the servo that you have not fine tuned using this equation:

$$\%\ \text{error} = \frac{\text{exact duration} - \text{predicted duration}}{\text{exact duration}} \times 100\%$$

The exact ***`period`*** is what you will arrive at by adjusting a single servo using trial and error. The predicted ***`period`*** is the value from the graph for that servo. You can expect percent errors of between 5 and 25% depending on the resolution of your graph and other factors such as imperfections in wheel alignment and slight differences in wheel size.

Chapter #2: Programming the Boe-Bot to Go Places

Chapter #2: Boe-Bot Navigation under Program Control

Chapter #2 is all about instructing the Boe-Bot where to go and how to get there. You'll write programs to make the Boe-Bot perform a variety of maneuvers. Some programs can be used for navigating tight spaces, others for drawing shapes. Whatever the maneuver, this chapter presents the tools for programming the Boe-Bot to perform it. Here's what you'll learn how to do in Chapter #2:

- Build a low battery indicator.
- Program your Boe-Bot to go a variety of directions, all in the same program.
- Write programs that fine tune the Boe-Bot's maneuvering skills.
- Write programs that remember long lists of movement instructions.
- Write programs that make the Boe-Bot accelerate and decelerate during maneuvers.

Along the way, Chapter #2 introduces a variety of PBASIC programming techniques, such as **`for...next`** loops and **`if...then`** statements. The exercises in this chapter also offer lots of practice in using variables and flow control to accomplish a variety of tasks. Some essential math for converting program commands into distance and speed are also introduced. For some, this will be a first glimpse into elementary Dynamics.

Converting Instructions to Motion

In the previous chapter, you programmed the Boe-Bot to move forward, backward, and turn in place. Additionally, software calibration settings were determined for programming the Boe-Bot to move straight forward, straight backward, and to stop and stay still.

Each Boe-Bot navigation program in Chapter #1, Activity #6 focused on one direction. If the Boe-Bot was programmed to go forward, it had to be reprogrammed to go backward and reprogrammed again to turn in place. In this chapter, all the directions will be incorporated into a single program. By determining how many pulses it takes to make the Boe-Bot rotate a certain amount during a turn, you can program the Boe-Bot to perform a variety of more precise maneuvers. For example, the Boe-Bot can be programmed to draw a square, or a cross, or a triangle.

This level of programmed maneuverability is well and good, but programming long and involved lists can become a complicated problem. PBASIC also features a simple and efficient method of recording and accessing long lists of directions in the program memory. You'll notice that while the Boe-Bot is performing its programmed maneuvers that it comes to abrupt stops when it changes direction. Commands can also be

added to make the Boe-Bot decelerate into and accelerate out of direction changes. This will solve the abrupt stops and extend the life of the Boe-Bot's servos.

The Boe-Bot's behavior when the batteries go low can be mystifying. Low batteries are also not good for the servos or the BASIC Stamp. The first activity will guide you through the construction and testing of a low battery indicator.

Activity #1: Low Battery Indicator

When the voltage supply drops below the level a device needs to function properly, it's called brownout. The BASIC Stamp protects itself from brownouts 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 waits for this condition. When the voltage comes back, the BASIC Stamp has been "reset." In response to a reset, the BASIC Stamp behaves essentially the same as if you had unplugged the power and then plugged it back in. In either case, what happens is that the program the BASIC Stamp was running before the reset starts over again at its beginning.

One way to indicate resets is to include an unmistakable signal at the beginning of all the Boe-Bot's programs. The signal then occurs every time the power gets plugged in, but it also occurs every time a reset due to brownout conditions occurs. The most effective signal for this is a speaker that emits a tone each time the BASIC Stamp program runs from the beginning or resets. The schematic symbol and top view of the piezoelectric speaker is used for emitting the tones in this activity are shown in Figure 2.1.

Parts List

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

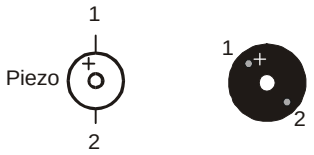


Figure 2.1: Piezo

Build It!

Figure 2.2 shows the piezospeaker and servo schematics along with a wiring diagram.

- ❑ Build the circuit as shown in Figure 2.2 (b).

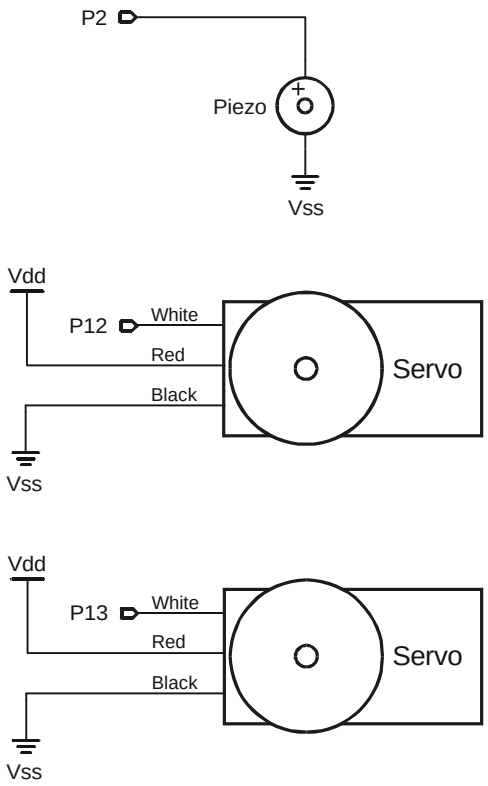
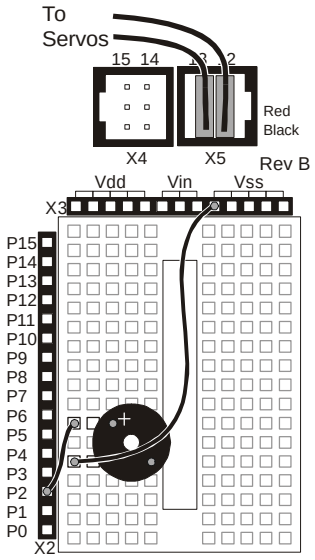


Figure 2.2 (a) Schematic



(b) Wiring diagram

✓ **TIPS**

New to building circuits from schematics? Check out Appendix F: Breadboarding Rules.

The instructions in all activities from this point forward will assume your servos are plugged in as shown in Figure 2.2 (a) and (b).

In Chapter 1, the millisecond and microsecond quantities were introduced. In this chapter, the hertz and kilohertz quantities are introduced. 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.

Hz and kHz

$$1\,Hz = \frac{1}{s} = 1\ s^{-1}$$

$$1\,kHz = \frac{1000}{s} = 1000\ s^{-1}$$

Programming the Low Battery Indicator

At the beginning of the program, a command that makes the speaker sound a tone will send the low battery signal. Then, the program listing should stay busy with an infinite loop until a reset occurs. If the circuit is correctly wired and the program is working, the tone should sound every time the power is unplugged then plugged back in. It should also sound every time the reset button is pressed. This guarantees that the tone will sound when a reset occurs due to low batteries.

- Connect the battery pack's barrel plug into the BOE's barrel jack.
- Connect the computer's serial cable to the BOE's DB9 connector.
- Enter Program Listing 2.1 into the Stamp Editor
- Run it by clicking Run then selecting Run.
- This program makes use of the Debug Terminal, so leave the serial cable connected to the BOE while Program Listing 2.1 is running.

```
' Robotics! v1.5, Program Listing 2.1, The Battery Indicator.
' {$Stamp bs2}                          ' Stamp Directive.

debug cls, "Beep!!!"                    ' Display while speaker beeps.
output 2
freqout 2, 2000, 3000                   ' Send a 3 kHz signal for 2 s.

loop:                                   ' Loop label.
  debug "Waiting for reset…", cr        ' Display while the BS2 is waiting.
goto loop:                              ' Go to the loop label (infinite loop).
```

- Verify that the low battery indicator works by pressing and releasing the reset button on the BOE.

Just as when the program first ran, the speaker should play a high pitched tone for 2 s. At the same time, the Debug Terminal should display the "Beep!!!" message. Then, all should go silent while the Debug Terminal displays line after line of the "Waiting for reset…" message.

- If the speaker did not play a tone, check your wiring and code, then try running the program again.

Debug display problems are typically caused by typing mistakes.

- If the Debug display does not behave as expected, check your program and make sure the code matches the example code shown in Program Listing 2.1 above.

How the Low Battery Indicator Program Works

- Use Appendix B or the BASIC Stamp Manual to look up the **`freqout`** and **`output`** commands introduced in program listing 2.1.

Program listing 2.1 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. Playing the tone actually takes two steps. First, P2 must be set to output using the **`output 2`** command. The audible tone is then played when the **`freqout 2, 2000, 3000`** command is executed. It sends pulses out P2 that make the piezoelectric speaker vibrate at 3 kHz for 2 s. When the tone is done, the program enters an infinite loop, displaying the same "Waiting for reset…" message over and over again. Each time the reset button on the BOE is pressed or the power is disconnected and reconnected, the Program starts over again.

The lines of code in the battery indicator program that generates the tone 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.

Your Turn

- Copy the **`freqout`** command from Program Listing 2.1 to the beginning of Program Listing 1.5 from Chapter #1, Activity #6.

- Run the modified version of Program Listing 1.5. The Boe-Bot should remain still and beep for two seconds before starting to move forward.

Activity #2: Controlling Distance

Up to now, Boe-Bot programs have featured infinite loops. For example, Program Listing 1.5 made the Boe-Bot move forward, but that's all it did. The Boe-Bot just kept going forward. This activity introduces a technique for controlling the distance the Boe-Bot travels.

Programming for Distance Control

An infinite loop can do a job, but it doesn't know when to stop. The best way to fix the problem is to replace the infinite loop with another kind of loop called a **`for…next`** loop. You can use a **`for...next`** loop to specify how many times the commands inside the loop are executed.

A **`for…next`** loop uses a variable to keep track of how many times the commands inside it are executed. A for…next loop makes use of a variable to store a number that gets changed each time through the loop. In PBASIC, a variable has to be declared before it can be used. Program Listing 2.2 shows an example of a variable declaration and an example of a **`for…next`** loop. The **`for…next`** loop is used instead of the infinite loop used in Program Listing 1.5. The **`for...next`** loop makes the Boe-Bot go forward and then stop by controlling the number of pulses sent to the servos.

- ❑ Connect the battery pack's barrel plug into the BOE's barrel jack.
- ❑ Connect the computer's serial cable to the BOE's DB9 connector.
- ❑ Enter Program Listing 2.2 into the Stamp Editor.
- ❑ Run it by clicking Run then selecting Run in the Stamp Editor.
- ❑ When the Boe-Bot speaker starts to beep indicating that the program is starting, press and hold the Reset button on the BOE.
- ❑ Unplug the serial cable from Boe-Bot.
- ❑ Place the Boe-Bot in the area you want it to navigate.
- ❑ Release the Reset button.
- ❑ Observe the Boe-Bot's behavior.

FYI	From this point forward, the procedure just described should be repeated for each program listing.

```
' Robotics! v1.5, Program Listing 2.2: Controlling Distance
' {$Stamp bs2}                          ' Stamp Directive.

'-----Declarations-------

  pulse_count   var   word              ' Declare a variable for counting.

'-----Initialization-----

  output 2                              ' Set P2 to output.
  freqout 2, 2000, 3000                 ' Signal program is starting/restarting.
  low 12                                ' Set P12 and 13 to output-low.
  low 13

'-----Main Routine-------

main:

  forward:
                          ' Forward routine.
    for pulse_count = 1 to 100          ' Loop that sends 100 forward pulses.
      pulsout 12, 500                   ' 1.0 ms pulse to right servo.
      pulsout 13, 1000                  ' 2.0 ms pulse to left servo.
      pause 20                          ' Pause for 20 ms.
    next

  stop                                  ' Stop until reset.
```

- Remember to unplug the battery pack from the BOE when you're not using it.

How the Controlling Distance Program Works

- Look up the **`for…next`** and **`stop`** commands in Appendix C: PBASIC Quick Reference or in the BASIC Stamp Manual before continuing.

PBASIC example programs in this text will be organized by section. Three of the five sections commonly used in a program are featured here: declarations, initialization, and main routine. As the PBASIC programs get longer and more involved, using comments to denote sections such as **`'-----Declarations-------`** and **`'-----Initialization-----`** makes the program considerably easier to read. The sections and their contents will be discussed in more detail after Program Listing 2.6.

A variable named **pulse_count** is declared to be a word's worth of variable storage space using the command **pulse_count var word**. A word variable can store numbers between 0 and 65535. The handy thing about a variable is that there are a variety of PBASIC commands that can be used to change the variable's value. Other PBASIC commands can use a variable's value to make decisions. Variables can also be used instead of numbers as arguments to certain PBASIC commands. You'll see examples of all these uses of variables in this chapter.

Table 2.1 shows the other options for variable declarations and the number ranges they can store. The size of the number a variable can store depends on how many bits it contains. A bit is a single, binary memory location that can either store a "1" or a "0." The more bits the larger the binary number you can store.

Table 2.1: Variable Declaration Sizes

Size Declaration	Number of Bits	Can Store Numbers Ranging from-to
bit	1	0 to 1
nib	4	0 to 15
byte	8	0 to 255
word	16	0 to 65535 (or –32768 to +32767)

The four commands in the initialization routine should all be pretty familiar by now. The first two are the **output** and **freqout** commands used to signal when the program starts running. The initialization routine also includes the **low** commands that are used to set the initial output values of the I/O lines used to control the servos.

Remember:	If the tone plays for no apparent reason when the Boe-Bot is in the middle of a maneuver, it indicates a brownout condition caused by low batteries.

The **main** routine uses a **for...next** loop followed by a **stop** command. The **for...next** loop replaces the infinite **goto** loop used in previous examples. The commands nested in the **for...next** loop should be familiar by now. They are the commands that send the pulses to the Boe-Bot servos. The values of the ***period*** arguments used in the **pulsout** commands are the values for making the Boe-Bot roll forward.

The first time through the for next loop, the value of **pulse_count** is set to "1." Then the two **pulsout** commands and one **pause** command are executed. When the program gets to the **next** statement, it jumps back up to the **for** statement. The second time through the loop, the **for** statement adds one to the value of **pulse_count**. Now the value of the **pulse_count** variable is "2," and the **for** statement checks to see if **pulse_count** is equal to 100 yet. Since **pulse_count** is not yet 100, the program continues to the next line. The three commands to pulse the servos and pause for 20 ms are executed again, and the **next** statement

sends program control back to the for statement again. The value of **pulse_count** is incremented again and compared to the upper limit, and so on.

The 100th time the program gets to the **next** statement in the **for...next** loop, it sends the program up to the **for** statement again. This time, the value of **pulse_count** is incremented to 101. Now, when **pulse_count** is compared to the value of the ***end*** argument, it is greater than the upper limit of 100. So instead of continuing to the commands that send another pulse, the **for** statement sends program control to the command immediately after the **next** statement.

The **stop** command is executed immediately after the **for...next** loop, and it makes the program **stop**. The only thing that can get the BASIC Stamp out of a **stop** command is a hardware reset. In other words, if you want to run the program again, press the Reset (Rst) button on the BOE.

Your Turn

- Modify the main routine of Program Listing 2.2 as shown below.

```
'-----Main Routine-------

main:                                   ' Main routine.

  forward:                              ' Forward routine.
    for pulse_count = 1 to 75           ' Send 75 forward pulses.
      pulsout 12, 500                   ' 1.0 ms pulse to right servo.
      pulsout 13, 1000                  ' 2.0 ms pulse to left servo.
      pause 20                          ' Pause for 20 ms.
    next

    pause 500                           ' Pause for 0.5 s.

  backward:
    for pulse_count = 1 to 75           ' Send 75 backward pulses.
      pulsout 12, 1000                  ' 2.0 ms pulse to right servo.
      pulsout 13, 500                   ' 1.0 ms pulse to left servo.
      pause 20                          ' Pause for 20 ms.
    next

    pause 500                           ' Pause for 0.5 s.

  stop                                  ' Stop until reset.
```

- Run the second modified version of Program Listing 2.2 again. The modified version of Program Listing 2.2 should make your Boe-Bot move forward, then backward, then stop.

When programming the Boe-Bot, the goal is often to make it move a specific distance or to execute a particular turn. It is helpful to know how to figure out how far the Boe-Bot will travel or turn when it is given a specific command. Circumference is equal to pi (π) multiplied by the wheel diameter:

circumference = π x wheel diameter

circumference = 3.14159 x 6.67 cm $\cong$ 21 cm

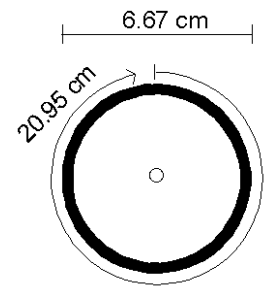


Figure 2.3: Wheel Diameter and Circumference

So, now we know that with one complete turn of the wheels, the Boe-Bot travels about 21 cm.

If we send pulses to the servo for the correct amount of time, the Boe-Bot can be made to travel a specific distance. This is because we can measure the speed that the Boe-Bot wheels turn. Once the speed is known, speed multiplied by time gives you distance. For example, with a **`pulsout`** command of 1000 delivered every 20 ms, the servo will turn at about 37.5 revolutions per minute (RPM), or 0.625 revolutions/sec. So the speed will be about:

21 cm/revolution x 0.625 revolutions/sec = 13.125 cm/s

The time it takes to make the Boe-Bot travel 50 cm is:

t_{travel} = 50 cm $\div$ 13.125 cm/s $\cong$ about 3.81 seconds

Since the pulse and pause periods are known, the time it takes for a single loop can be calculated:

t_{loop} = 1.0 ms + 2.0 ms + 20 ms = 23 ms

Calculating the number of loops (the ***end*** argument for the **`for...next`** loop) is simply a matter of dividing t_{loop} into t_{travel}.

Number of loops = 3.81 sec $\div$ 23 ms/loop

= 3.81 s $\div$ 0.023 s/loop $\cong$ 166 loops

- Modify Program Listing 2.2 for 166 forward loops (50 cm), then run it.

How far did the Boe-Bot go? Several factors can affect how far it moves, including differences between servos and battery voltage. However, the estimate of 166 is a good initial value to try. The **`for...next`** loop's ***end*** argument can then be fine tuned.

Activity #3: Maneuvers – Making Turns

If the same value is added to the center pulse width of one servo and subtracted from the center pulse width of the other, the Boe-Bot will travel in a straight line, either forward or backward. When the right servo gets a **`pulsout`** ***`period`*** of 500 (1.0 ms) and the left servo gets a **`pulsout`** ***`period`*** of 1000 (2.0 ms), the Boe-Bot goes forward. When the pulse periods for each servo are swapped, the Boe-Bot goes backward.

If both servos receive 1.0 ms pulses, they turn in the same direction and cause the Boe-Bot to rotate counterclockwise. If many pulses are applied, the Boe-Bot will keep rotating. If 35 or so pulses are applied, the net effect is in the neighborhood of a 90° left turn. The same principles apply if both servos receive 2.0 ms pulses, except that the Boe-Bot will rotate clockwise instead of counterclockwise.

Programming Left and Right Turns

Program Listing 2.3 illustrates how the **`forward`** and **`backward`** routines from Program Listing 2.2 can be modified to make the Boe-Bot execute turns.

- Enter and run Program Listing 2.3.

```
' Robotics! v1.5, Program Listing 2.3: Turning in place.
' {$Stamp bs2}                          ' Stamp Directive.

'-----Declarations-------

  pulse_count   var   word               ' Declare a word size loop counter.

'-----Initialization-----

  output 2                               ' Set P2 to output.
  freqout 2, 2000, 3000                  ' Send a 3 kHz signal for 2 s.
  low 12                                 ' Set P12 and 13 to output-low.
  low 13

'-----Main Routine-------

main:                                    ' Main routine
```

```
left_turn:                              ' Left turn routine.
  for pulse_count = 1 to 35             ' Send 35 left rotate pulses.
    pulsout 12, 500                     ' 1.0 ms pulse to right servo.
    pulsout 13, 500                     ' 1.0 ms pulse to left servo.
    pause 20                            ' Pause for 20 ms.
  next

  pause 500                             ' Pause for 0.5 s.

right_turn:                             ' right turn routine.
  for pulse_count = 1 to 35             ' Send 35 left rotate pulses.
    pulsout 12, 1000                    ' 2.0 ms pulse to right servo.
    pulsout 13, 1000                    ' 2.0 ms pulse to left servo.
    pause 20                            ' Pause for 20 ms.
  next

  pause 500                             ' Pause for 0.5 ms.

stop                                    ' Stop until reset.
```

How the Turning in Place Program Works

Only three changes were made to Program Listing 2.2 in order to make Program Listing 2.3. First, the **`forward:`** and **`backward:`** labels were changed to **`left_turn:`** and **`right_turn:`** respectively. Then, the **`left_turn`** routine had both **`pulsout`** ***`period`*** arguments set to 500. The **`pulsout`** arguments in the **`right_turn`** routine were both set to 1000.

Your Turn

- Add the forward and backward routines from Program Listing 2.2 to Program Listing 2.3. To make the Boe-Bot go forward and backward before turning left and right, insert the **`forward`** and **`backward`** routines between the **`main:`** label and the **`left_turn:`** label. Run the program and observe the results.

- Instead of using an ***`end`*** argument of 35 for each turn **`for...next`** loop, try ***`end`*** arguments of 30, 31, 32,..., 39, 40. Record the value that makes the Boe-Bot do the most precise quarter turn. The values may or may not be the same for both **`left_turn`** and **`right_turn`** routines.

Activity #4: Maneuvers – Ramping

Ramping is a way to gradually increase the speed of the servos instead of suddenly making them go the opposite direction. This technique can increase the life expectancies both of your Boe-Bot's batteries and servos.

Programming for Ramping

The key to ramping is to use variables along with constants for servo pulse period. A **`for…next`** loop increments a variable each time the code nested between the **`for`** and **`next`** statements are executed. Since the value of this variable increases incrementally, it can be used to gradually increase the pulse widths. Program Listing 2.4 shows how this technique can be used to make the Boe-Bot ramp up to and back down from the speed used in the forward routine.

```
' Robotics! v1.5, Program Listing 2.4: Ramping for Start and Stop
' {$Stamp bs2}                          ' Stamp Directive.

'-----Declarations-------

  pulse_count  var   word               ' For...next loop counter.
  right_width  var   word               ' Variable stores right pulse width.
  left_width   var   word               ' Variable stores left pulse width.

'-----Initialization-----

  output 2                              ' Set P2 to output.
  freqout 2, 2000, 3000                 ' Signal program is starting/restarting.
  low 12                                ' Set P12 and 13 to output-low.
  low 13

'-----Main Routine-------

main:                                   ' Main routine.
  ramp_up_forward:                      ' Routine ramps into forward motion.
    for pulse_count = 0 to 250 step 2   ' For loop counts up in steps of 2.
      pulsout 12, 750 - pulse_count     ' Pulse sent is 1.5 ms – pulse_count value.
      pulsout 13, 750 + pulse_count     ' Pulse sent is 1.5 ms + pulse_count value.
      pause 20                          ' Pause for 20 ms.
    next

  forward:                              ' Forward routine.
    for pulse_count = 1 to 100          ' Loop that sends 100 forward pulses.
      pulsout 12, 500                   ' 1.0 ms pulse to right servo.
      pulsout 13, 1000                  ' 2.0 ms pulse to left servo.
```

```
    pause 20                          ' Pause for 20 ms.
  next

ramp_down_forward:                    ' Routine ramps out of forward motion.
  for pulse_count = 250 to 0 step 2   ' For loop counts down in steps of 2.
    pulsout 12, 750 - pulse_count     ' Pulse sent is 1.5 ms – pulse_count value.
    pulsout 13,750 + pulse_count      ' Pulse sent is 1.5 ms + pulse_count value.
    pause 20                          ' Pause for 20 ms.
  next

stop                                  ' Stop until reset.
```

How the Ramping Program Works

The pulses in the **ramp_up_forward** routine are nested in a **for…next** loop. Note the ***step 2*** argument added to the **for** statement. None of the previous **for...next** loop examples used this argument, so the value of **pulse_count** was incremented in steps of 1. The **step 2** argument in the **for** statement makes the value of **pulse_count** increment in steps of 2.

```
ramp_up_forward:
  for pulse_count = 0 to 250 step 2
    pulsout 12, 750 - pulse_count
    pulsout 13, 750 + pulse_count
    pause 20
  next
```

The first time though the **for...next** loop, the value of **pulse_count** is "0." The second time through, **pulse_count** is incremented to "2," and the third time through it's "4," and so on, up to 250. The key to ramping is to modify the pulse period a little each time a pulse is sent to the servo until it's up to the desired period. Incorporating the value of **pulse_count** into the ***period*** argument of each **pulsout** command can be used to incrementally increase the pulse period in this way.

Let's take a look at the pulses sent to the right servo. The command for sending the pulse is **pulsout 12, 750 – pulse_count**. Each time through the **for...next** loop, the value of **pulse_count** increases by two. This means that each time through the loop, the value of the **pulsout** command's ***period*** argument is decreased by two, since **pulse_count** is subtracted from 750. By the last pass through the loop, the value of **pulse_count** is up to 250, which is subtracted from 750 to give a value of 500. For the left servo, the value of **pulse_count** is added to 750, and the last time through the loop gives a ***period*** of 1000. When these target values are reached, the servos are up to full speed forward, and the program can move on to the familiar **forward** routine.

When the **forward** routine is finished, the challenge is to ramp the pulse widths back down to the resting value of 750. A useful feature of PBASIC is that **for...next** loops count down instead of up if the ***start*** argument is larger than the ***end*** argument. This method was used for counting back down in the **ramp_down_forward** routine. The **for…next** loop started **pulse_count** at a value of 250, then counted down to 0 in steps of 2. This is because the command **for pulse_count = 100 to 0 step 2** was used.

Your Turn

- ❑ Develop routines that ramp into and back out of the **backward**, **right_turn** and **left_turn** routines developed in Activity #2 and #3.

It will take some fine tuning to get the Boe-Bot to do 90° and 180° turns. Adjusting **for** statement's ***step***, ***start*** and ***end*** arguments will help make the turns more accurate. For the 90° turns, it's also possible to ramp into and back out of the turns without ever reaching full speed (pulse width).

Activity #5: Remembering Long Lists Using EEPROM

The BASIC Stamp stores its tokenized version of the PBASIC program in electrically erasable programmable read only memory (EEPROM). Physically, the EEPROM is the small black chip on the BASIC Stamp II module labeled "24LC16B." This part is made by Microchip Inc. 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).

FYI	If the data collides with your program, the PBASIC program won't execute properly.

EEPROM memory is different from RAM variable storage in several aspects:

- 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 is stored in leftover space.

You can view the contents of the BASIC Stamp's EEPROM by clicking Run and selecting Memory Map. Figure 2.4 shows the memory map for Program Listing 2.3. Note the Condensed EEPROM Map in the lower middle of the figure. The narrow stripe across the bottom represents the small portion of EEPROM that Program Listing 2.3 occupies.

This program might have seemed large while you were typing it in, but it only takes up 128 of the available 2048 bytes of available program memory. There currently is enough room for quite a long list of instructions. One of the simpler approaches is to store characters indicating which way to go. Since a character occupies a byte in memory, there is currently room for 1920 one-character direction instructions.

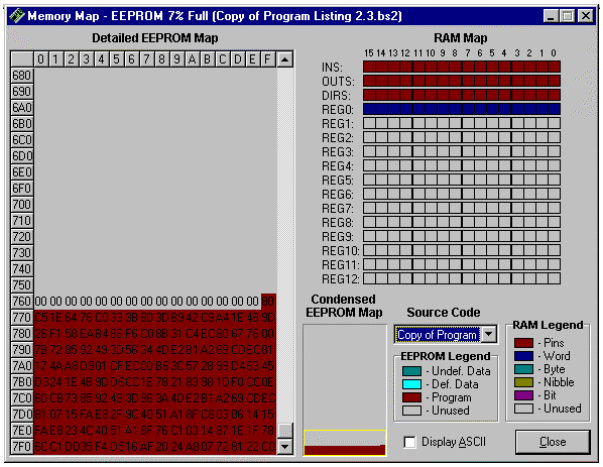


Figure 2.4: BASIC Stamp memory map.

EEPROM Navigation

The **`data`** directive is usually placed in the **`declarations`** section of a PBASIC program, and it's the best way to store long lists. Program Listing 2.5 shows an example of how the **`data`** directive can be used to store lists of Boe-Bot navigation instructions.

```
' Robotics! v1.5, Program Listing 2.5: EEPROM Navigation
' {$Stamp bs2}                          ' Stamp Directive.

'----- Declarations --------------

' Label for the declarations routine.
  pulse_count   var   word               ' Declare a variable named pulse_count.
  EE_address   var   byte                ' Stores & increments EEPROM address.
  instruction  var   byte                ' Stores instruction retrieved from EEPROM.

  data         "FFFBBLFFRFFQ"            ' List of Boe-Bot navigation instructions.

'----- Initialization ------------

  output 2                               ' Set piezoelectric speaker line to output.
  freqout 2, 2000, 3000                  ' Send the program start/low battery tone.
  low 12                                 ' Set P12 to output-low.
  low 13                                 ' Set P13 to output-low.

'----- Main Routine --------------

main:                                    ' Main routine label.
```

```
  read EE_address, instruction          ' Read at EE_address & store in instruction
  EE_address = EE_address + 1           ' Increment EE_address for next read

  if instruction = "F" then forward     ' Check for forward command.
  if instruction = "B" then backward    ' Check for backward command.
  if instruction = "R" then right_turn  ' Check for right turn command.
  if instruction = "L" then left_turn   ' Check for left turn command.

  stop                                  ' Stop executing commands until reset

'----- Navigation Routines -------

  forward:                              ' Forward routine.
    for pulse_count = 1 to 75           ' Send 75 forward pulses.
      pulsout 12, 500                   ' 1.0 ms pulse to right servo.
      pulsout 13, 1000                  ' 2.0 ms pulse to left servo.
      pause 20                          ' Pause for 20 ms.
    next
  goto main                             ' Send program back to the main.

  backward:
    for pulse_count = 1 to 75           ' Send 75 backward pulses.
      pulsout 12, 1000                  ' 2.0 ms pulse to right servo.
      pulsout 13, 500                   ' 1.0 ms pulse to left servo.
      pause 20                          ' Pause for 20 ms.
    next
  goto main                             ' Send program back to the main.

  left_turn:                            ' Left turn routine.
    for pulse_count = 1 to 35           ' Send 35 left rotate pulses.
      pulsout 12, 500                   ' 1.0 ms pulse to right servo.
      pulsout 13, 500                   ' 1.0 ms pulse to left servo.
      pause 20                          ' Pause for 20 ms.
    next
  goto main                             ' Send program back to the main label.

  right_turn:                           ' right turn routine.
    for pulse_count = 1 to 35           ' Send 35 left rotate pulses.
      pulsout 12, 1000                  ' 2.0 ms pulse to right servo.
      pulsout 13, 1000                  ' 2.0 ms pulse to left servo.
      pause 20                          ' Pause for 20 ms.
    next
  goto main                             ' Send program back to the main .
```

How the EEPROM Navigation Program Works

- Look up the **data**, **read** and **if...then** commands in Appendix C: PBASIC Quick Reference or in the BASIC Stamp Manual before continuing.

Program Listing 2.5 introduces two new variables. Instead of being words, both of them are bytes, which means they can store numbers between zero and 255. The first new variable is **EE_address**, which is used for specifying the EEPROM address to **read** a direction instruction from EEPROM. The second new variable is named **instruction**. This variable is used to store the instruction character read from EEPROM.

```
declarations:
  pulse_count  var   word
  EE_address   var   byte
  instruction  var   byte
```

The next declaration is the actual data to be stored in EEPROM. This data is stored as a string of characters. When these characters are stored, they are stored as numbers that correspond to the letters you see in the **data** directive. In the Your Turn section of this activity, you'll view the memory map and look at both the characters and their numeric representations in EEPROM.

```
  data         "FFFBBLFFRFFQ"       ' Movement data
```

Although the initialization routine is the same one used in Program Listing 2.3, the main routine is not. The main routine first reads EEPROM address 0, and stores it in the **instruction** variable. Then, **EE_address** is incremented so that the next read cycle will look at address 1. A series of **if...then** statements is used to decide what to do based on the character retrieved from EEPROM and stored in the **instruction** variable. The **if...then** statements check to see if it's one of four known instruction characters: "F," "B," "R," and "L." For example, if the character is an "R," the first two **if...then** statements are skipped because neither of them is true. Since the third **if...then** statement is true, the program skips to the **right_turn** routine.

```
main:
  read EE_address, instruction
  EE_address = EE_address + 1

  if instruction = "F" then forward
  if instruction = "B" then backward
  if instruction = "R" then right_turn
  if instruction = "L" then left_turn

  stop
```

When the program gets to the **`right_turn`** routine, it executes 35 right turn pulses. Then the **`goto main`** command sends the program back to the **`main`** routine.

```
right_turn:
  for pulse_count = 1 to 35
    pulsout 12, 1000
    pulsout 13, 1000
    pause 20
  next
goto main
```

When the program gets back to the **`main`** routine, the next EEPROM instruction is fetched, and the instruction is checked by the four **`if...then`** statements again. The process is repeated until the quit character "Q" is read from EEPROM. When "Q" is loaded into the **`instruction`** variable, it fails all four **`if...then`** tests. So, the program does not go to any of the navigation routines. Instead, the program goes to the command that follows the series of **`if...then`** statements, which is the **`stop`** command.

Your Turn

- With Program Listing 2.5 active in the 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 2.5. The numbers shown are the hexadecimal ASCII (American Standard Code for Information Interchange) codes that correspond to the letters you entered in your data statement.

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

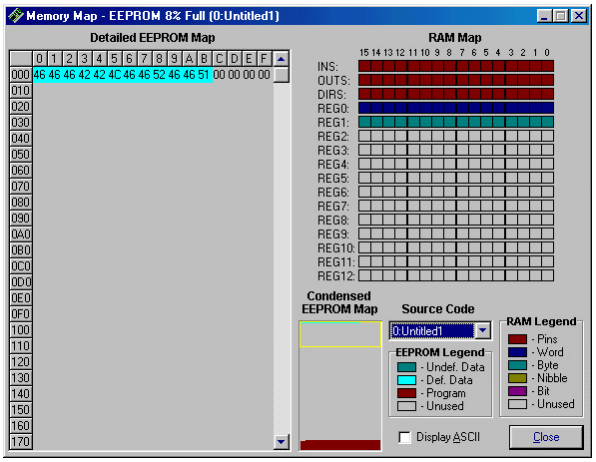


Figure 2.5: Memory Map with stored instructions visible in Detailed EEPROM Map.

Now the direction instructions will appear in a more familiar format shown in Figure 2.6. Instead of ASCII codes, they appear as the actual characters you recorded using the **`data`** directive.

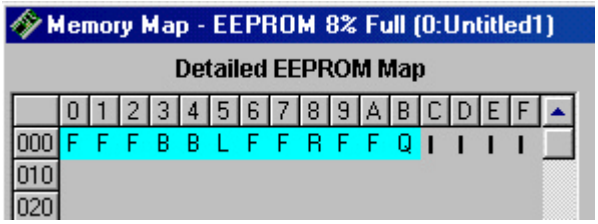


Figure 2.6: Close-up of the detailed EEPROM Map after Display ASCII box is checked.

In its current form, Program Listing 2.5 can only access up to 256 characters. If the **`EE_address`** variable is re-declared to be a word variable, you can address more locations than the EEPROM actually contains. 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. Remember that the last character in the **`data`** directive should always be a "Q."

- 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.

Activity #6: Simplify Navigation with Subroutines

All the navigation routines in Activity #5 ended with **`goto`** commands. Each **`goto`** command sent program control to the **`main:`** label. A similar technique involves the use of subroutines, and several example programs in this text will make use of them. Since the example program in Activity #7 makes use of subroutines, let's take a look at how one works before moving on.

Programming Navigation with Subroutines

A subroutine is a segment of code that does a particular job. To make the subroutine do its job, a command is used in the main routine that "calls" the subroutine. The command for calling a subroutine is the **`gosub`** command, and it's similar to the **`goto`** command. A **`goto`** command tells the program to go to a label, and then start executing instructions. The **`gosub`** command tells the program to go to a label, and start executing instructions, but come back when finished. A subroutine is finished when the **`return`** command is encountered.

Program Listing 2.6 uses a different technique for navigation. It sets variable values for number of pulses delivered and servo pulse widths. After the variable values have been set, the main program calls a subroutine that makes use of the information stored in the variables to pulse the servos.

```
' Robotics! v1.5, Program Listing 2.6: Subroutine Navigation.
' {$Stamp bs2}                          ' Stamp Directive.

'----- Declarations --------------

loop_count   var   word                 ' For...next loop counter.
right_width  var   word                 ' Variable stores right pulse width.
left_width   var   word                 ' Variable stores left pulse width.
pulse_count  var   byte                 ' Used to set number of pulses delivered.

'----- Initialization ------------

output 2                                ' Set P2 to output.
freqout 2, 2000, 3000                   ' Signal program is starting/restarting.
low 12                                  ' Set P12 and 13 to output-low.
low 13

'----- Main Routine --------------

main:                                   ' Main routine.

  forward:                              ' Forward routine.
    pulse_count = 75                    ' Set pulse count for 75 pulses.
    right_width = 500                   ' Set right pulse width to 1.0 ms.
    left_width = 1000                   ' Set left pulse width to 2.0 ms.
    gosub pulses                        ' Call the pulses subroutine.
    pause 500                           ' Pause for 0.5 s.

  backward:                             ' Backward routine.
    pulse_count = 75                    ' Set pulse count for 75 pulses.
    right_width = 1000                  ' Set right pulse width to 2.0 ms.
    left_width = 500                    ' Set left pulse width to 1.0 ms.
    gosub pulses                        ' Call the pulses subroutine.
    pause 500                           ' Pause for 0.5 s.

goto main                               ' Go to main label (infinite loop).

'----- Subroutines ---------------

pulses:                                 ' Pulses subroutine.
  for loop_count = 1 to pulse_count     ' Use pulse_count for number of pulses.
    pulsout 12, right_width             ' Use right_width for right pulse width.
```

```
    pulsout 13, left_width              ' Use left_width for left pulse width.
    pause 20                            ' Pause 20 ms.
  next
return                                  ' Return from subroutine.
```

How the Subroutine Navigation Program Works

- Look up the **gosub** and **return** commands in Appendix C: PBASIC Quick Reference or in the BASIC Stamp Manual before continuing.

Figure 2.7 shows how the **backward** routine calls the **pulses** subroutine. The commands in the **pulses** subroutine are executed until the program gets to the **return** command. The **return** command sends the program back to the command just after **gosub pulses** command, which is **pause 500** in this case. This same process occurred in the **forward** routine, which was executed before the **backward** routine shown.

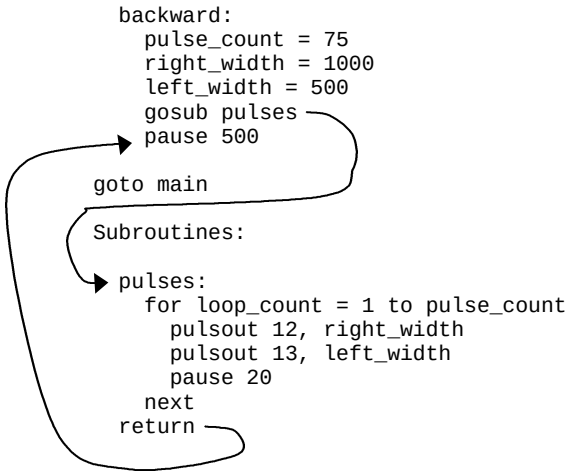


Figure 2.7: Program flow for subroutines.

Your Turn

- Add routines to the main routine that set the values for **right_turn** and **left_turn**.

Activity #7: All Together Now

This chapter has introduced a variety of navigation techniques:

- navigation routines that control distance and turning
- speed ramping
- remembering long lists
- calling subroutines
- using variables to set values used by subroutines

All these elements now can be tied together to make a more robust navigational program.

Programming EEPROM Navigation with Ramping

The **data** directive in Program Listing 2.7 records two different things in EEPROM for each maneuver: a direction and a number of pulses to deliver. Both values are stored in variables. The subroutine that delivers the pulses to the servos does automatic ramping into and out of each maneuver.

```
' Robotics! v1.5, Program Listing 2.7: EEPROM Navigation with Ramping
' of navigational instructions.
' {$Stamp bs2}                          ' Stamp Directive.

'----- Declarations --------------

  pulse_count  var   byte               ' Used to set number of pulses delivered.
  loop_count   var   byte               ' For...next loop counter.
  right_width  var   word               ' Variable stores right pulse width.
  left_width   var   word               ' Variable stores left pulse width.
  old_right    var   word               ' Stores previous right pulse width.
  old_left     var   word               ' Stores previous left pulse width.
  right_gap    var   word               ' Difference btwn target and actual width.
  left_gap     var   word               ' Difference btwn target and actual width.
  EE_address   var   byte               ' Stores & increments EEPROM address.
  instruction  var   byte               ' Stores instruction retrieved from EEPROM.
  direction    var   byte               ' Stores number from data table for branch.

  data         "F", 125, "B", 125, "R", 75, "L", 100, "F", 150, "Q"

'----- Initialization ------------

  right_width = 750                     ' Set initial values for variables.
  left_width = 750
  old_right = 750
  old_left = 750
  pulse_count = 0
  output 2                              ' Set P2 to output.
  freqout 2, 2000, 3000                 ' Signal program is starting/restarting.
  low 12                                ' Set P12 and 13 to output-low.
  low 13

'----- Main Routine --------------

main:                                   ' Main routine.

  gosub pulses                          ' Call the pulses subroutine.

  read EE_address, instruction          ' Read the first EEPROM instruction.
  read EE_address + 1, pulse_count      ' Read the second EEPROM instruction.
```

```
  EE_address = EE_address + 2           ' Increment EE_address by 2.

  ' The lookdown instruction is used together with the branch instruction
  ' to direct the program to the appropriate routine for setting pulse widths.
  ' After the variable values are set, program control is returned to main.
  ' The first command in the main routine calls the pulses subroutine.

  lookdown instruction, = ["B","L","R","F","Q"], direction
  branch direction,[backward,left_turn,right_turn,forward,quit]

      backward:     right_width = 1000: left_width = 500: goto main
      left_turn:    right_width = 500: left_width = 500: goto main
      right_turn:   right_width = 1000: left_width = 1000: goto main
      forward:      right_width = 500: left_width = 1000: goto main

      quit:         stop

'----- Subroutines ---------------

  ' The target pulse values of left_width and right_width are compared to
  ' their previous values.  The right_gap and left_gap variables are used to
  ' determine the pulse widths.  See the How EEPROM Navigation with Ramping
  ' Works section for further explanation.

  pulses:
    right_gap = right_width - old_right
    left_gap =  left_width - old_left

    for loop_count = 1 to pulse_count

      right_gap = right_gap - 5 + (10*right_gap.bit15)
      left_gap = left_gap - 5 + (10*left_gap.bit15)

      pulsout 12, right_width - right_gap
      pulsout 13, left_width - left_gap

      pause 20

    next

    old_right = right_width - right_gap
    old_left = left_width - left_gap

  return
```

Advanced Topic: How EEPROM Navigation with Ramping Works

- Look up each of the following new commands in Appendix C: PBASIC Quick Reference or in the BASIC Stamp Manual before continuing: **lookdown, branch.**

Six word-size variables are added to the declarations section of Program Listing 2.5. The variables **old_left** and **old_right** are for remembering the previous pulse period for each servo. The variables **right_gap** and **left_gap** are variables that change slightly each time a pulse is applied. When the gap is large, the pulse width is closer to the old pulse values. When the gap is small, the pulses are closer to the new pulse periods.

The **data** directive has been restructured so that it stores a direction, followed by a number of pulses. This way, the Boe-Bot can be given directions like: go forward 125 pulses, then backward 125 pulses, then right 75 pulses, and so on.

Initial values are set for the variables that keep track of pulse widths. They all start with the center value of 750. The first command in the **main** routine is **gosub pulses**. Since the initial number of pulses to deliver is a value of "0," and the initial values of the old and current servo instructions are at center, the pulses routine will do nothing the first time through. Every time through the **main** routine after that, the **gosub** command calls the **pulses** subroutine, which pulses the servos with values determined from the previous pass through the **main** routine.

There are now two **read** commands in the **main** routine instead of just one. This is so that both the direction and number of pulses can be read. The first **read** command still accesses a letter that denotes a Boe-Bot direction. The second **read** command looks up **EE_address + 1**, which is the location of the number that follows the direction in the **data** statement. This is a handy way to access the pulse count value in a single command. Then **EE_address** is incremented by two instead of one. This way, the next time the first **read** command looks for a letter, it will skip over the number (of pulses) and find the next letter in the sequence.

Now that a letter indicating direction has been stored in the **instruction** variable and the number of pulses have been stored in the **pulse_count** variable, it's time for the BASIC Stamp to figure out what that letter stored in **instruction** means. Instead of using a series of **if...then** statements, two commands are used. The first is the **lookdown** command. The **lookdown** command will take the value "B" from **instruction** and find that it's the first in the list. Actually, as far as the **lookdown** command is concerned, it's really the zero entry in the list since the **lookup** command starts counting from zero. So, it stores the number 0 in the **direction** variable. If **instruction** is an "L," the number 1 will be stored in **direction**. If **instruction** is "R," the number 2 will be stored in **direction**, and so on.

The **branch** command skips to a label based on the variable's value. If **direction** is "0," **branch** will skip to the **backward** routine. If **direction** is "1," **branch** will skip to the **left_turn** label. If **direction** is "2," branch will skip to the **right_turn** label, and so on.

You may not have been aware that you could place more than one command separated by colons on a line. The **backward**, **left_turn**, **right_turn**, **forward**, and **quit** routines each occupy a single line. The only problem that can occur when multiple PBASIC commands are on a single line is that a label cannot be found in the middle of a line, only at the beginning. Each of the **direction** routines sets the value of **right_width** and **left_width** ***period*** values, just as in previous programs. Instead of calling the pulses routine, each of the four direction routines ends with a **goto main** command. Note that as soon as the program gets to **main,** it calls the **pulses** subroutine.

The **pulses** subroutine first determines the difference between the previous value sent to it for each servo and the current value. Taking the right servo as an example, the variable that stores this value is **right_gap**, because it stores the current **right_width** minus the **old_right** width. The process is the same for the left servo, except the sign is reversed. Next, the **for...next** loop applying **pulse_count** pulses is started.

Assuming that **right_width** is different from **old_right**, the **right_gap** value will start large. Since **right_gap** is subtracted from **right_width** when the **pulsout** command is executed, it stands to reason that the **right_gap** value should initially be large. Then, with each pulse, the **right_gap** value should get smaller and smaller as the pulse width ramps up to its target value. A single command takes care of decrementing **right_gap** if it's positive and incrementing it if it's negative.

```
right_gap = right_gap - 5 + (10*right_gap.bit15)
```

How does this expression work? If **right_gap** is a positive number, right_gap.bit15 will be zero. So, if **right_gap** is positive, it still gets 5 subtracted from it. However, if **right_gap** is negative, right_gap.bit15 is "1," not "0." This is because of how the two's complement binary system works in the BASIC Stamp. This means that **right_gap** also gets 10 added to it for a net gain of 5. The same principle applies for the **left_gap**.

Next the pulses are delivered. Note that each ***period*** argument is an expression. For the **pulsout** command to P12, the ***period*** is **right_width – right_gap**, and for the ***period*** delivered to P13, it's **left_width – left_gap**.

It's important to keep track of what very last pulse values delivered to the servos were before fetching the next navigation instruction from EEPROM. The two commands after the **for...next** loop that delivers pulses save the pulse widths.

```
old_right = right_width - right_gap
old_left = left_width - left_gap
```

The next time the program returns to the **pulses** subroutine, the first two commands set the initial values:

```
pulses:
  right_gap = right_width-old_right
  left_gap =  left_width - old_left
```

The **for...next** loop works in the pulses subroutine gradually adjusts the pulse values from where they started to the target value.

Your Turn

- Modify the **data** statement to make the Boe-Bot draw a square with one-meter sides. You can add a second **data** statement right below the first if you need more instructions. Just remember that there should only be one "Q" at the very end of the last **data** statement.

Summary and Applications

A low battery indicator circuit was built and tested using a BASIC Stamp program. This chapter also introduced a variety of PBASIC programs to make the Boe-Bot execute different maneuvers. Examples include controlling the Boe-Bot's distance and turns, along with methods for programming the Boe-Bot to travel measured distances. Examples of speed ramping, EEPROM navigation as well as an example of integrating the navigation algorithms introduced in this chapter also were provided.

Real World Example

When the Boe-Bot's batteries go below a certain level, the Boe-Bot sends a signal indicating that something went wrong by sending electronic pulses to a piezoelectric speaker. These speakers are common. Think of the number of appliances that beep when you press a key or do something. Your microwave oven, grocery store cash registers, and alarm system keypads all have this feature. It should come as no surprise that each has a microcontroller monitoring the keys on a keypad. Level sensors are also common in industry, where voltage, pressure, temperature and a variety of other conditions have to be controlled. Microcontrollers are often used to monitor these conditions and make an alarm speaker sound when the process goes outside acceptable levels.

Micro-controlled motion is also all around us. Although there may not be that many autonomous rolling robots in your household yet, there are many other gizmos with micro-controlled moving parts. Printer heads and computer disk drives are two examples that use stepper motors. Servos controlled by microcontrollers are also used in a variety of places. Many automobile systems rely on servos to control small moving parts in various engine and emission systems. Industrial servos maintain many factory processes, often in conjunction with the level sensors discussed earlier.

Boe-Bot Applications

Programmed navigation is the foundation for a variety of other Boe-Bot activities. In the Projects section, you'll work on programming the Boe-Bot to navigate a variety of shapes and on fine tuning some of the navigational algorithms developed in this chapter. In Chapter #3, we'll use some of these routines to respond to sensor inputs. The navigation routines developed in this chapter will be especially helpful in getting around obstacles that are detected. One of the most popular occupations for autonomous robots is solving mazes, which are full of obstacles. Appendix I has the Boe-Bot competition rules, which feature a variety of obstacles that need to be navigated. Responding automatically to certain situations can increase the Boe-Bot's ability to navigate features within the maze. For example, when a corner is detected, instead of trying to navigate the corner based solely on sensor input, the Boe-Bot can rely on preprogrammed routines, stored in EEPROM.

Questions and Projects

Questions

1. How does the Boe-Bot's low battery indicator work?

2. Name and describe the three arguments needed to make a **`freqout`** command work.

3. Which argument in Program Listing 2.2 controls the distance of the **`forward`** routine?

4. What is the ***`period`*** of the **`pause`** before the **`stop`** in Program Listing 2.2?

5. What are the left and right servo pulse widths, in ms, for forward, backward, left turn and right turn?

6. What PBASIC **`for...next`** loop feature makes ramping back down as easy as ramping up in Program Listing 2.4?

7. What is the difference between a routine and a subroutine? What command is used to call a subroutine? What happens when a subroutine is finished? What command signifies that a subroutine is finished, and what does it do?

8. How many memory locations does the BASIC Stamp's EEPROM have? Where does program memory start in the EEPROM, and which way does it build as the program gets larger? Where does data storage start in the EEPROM, and which way does it build as more data is added? What command is used to retrieve data from the EEPROM?

9. What argument comes after the **`if`** in an **`if...then`** statement? What argument comes after the then?

10. How does the **`lookdown`** command in Program Listing 2.7 make the transition from the letters read from EEPROM to the numbers used in the **`branch`** command?

Exercises

1. Adjust the tone generated in Program Listing 2.1 so the frequency of the tone played by the piezoelectric speaker is 1.5 kHz with a ***period*** of three seconds.

2. Assume that your program has become quite large, and you need to conserve RAM. Change the **pulse_count** variable declaration in Program Listing 2.2 so that it requires half the RAM currently used (8-bits instead of 16).

3. Increase the **pause** ***period*** between left turn and right turn in Program Listing 2.3 to two seconds.

4. Modify Program Listing 2.4 so that it's ramping rate doubles.

5. Program Listing 2.5 currently accepts any letter other than "F," "B," "R," and "L" to end the program. Modify it so that it only accepts the letter "Q."

6. Modify Program Listing 2.6 so that it can travel up to 65534 pulses in a single subroutine execution?

7. Modify Program Listing 2.7 so that the **lookdown** command is no longer necessary. Hint: Store digits instead of letters for direction control.

Projects

One aspect of Boe-Bot navigation that was not covered in this chapter is drawing curves. A curve occurs any time the left wheel is not turning the same speed as the right wheel. Curves also occur when the left wheel is accelerating/decelerating at a different rate than the right wheel. Keep in mind that the number of pulses delivered is a way of controlling distance for a given pulse width (speed).

1. The Boe-Bot is being used to transport reactive material, in particular solid sodium and water. If the two react, they explode, leaving your Boe-Bot as a pile of components. In order to carefully transport the chemicals, you will need to start movements with gradually increasing velocity. Create a program that drives the Boe-Bot in a one-meter square with smooth turning transitions.

2. Create a simple movement pattern with several directions, for example F,B,R,R,F,F,L, and lastly F. These patterns would be stored and read from the EEPROM. When you are done executing the pattern, trace the same pattern backwards.

3. Create source code for the following movement patterns:

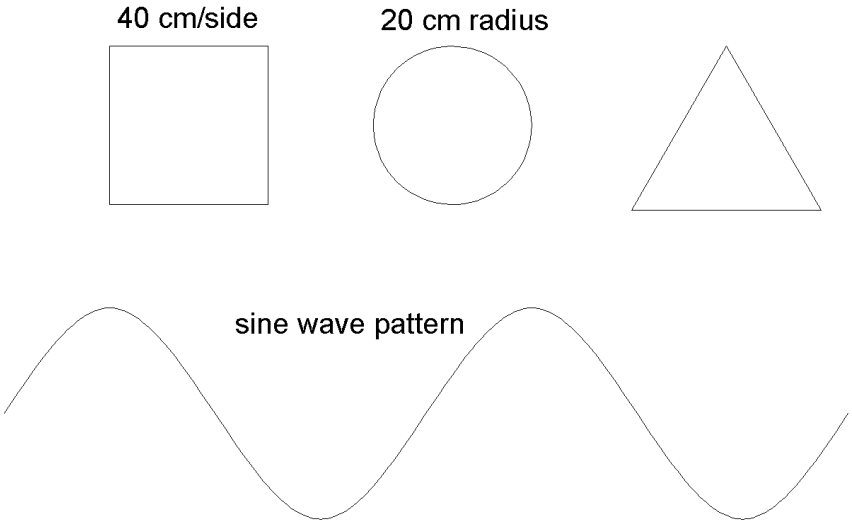


Figure 2.8: Boe-Bot paths to program.

Chapter #3: Tactile Navigation with Whiskers

Tactile Navigation

The Whiskers kit is so named because the kit's bumper switches look like whiskers, though some argue they look more like antennae. At any rate, whiskers give the Boe-Bot the ability to sense the world around it with tactile inputs. The Boe-Bot can use these whiskers to navigate only by touch. Although the activities in this chapter focus on using just the whiskers, they can also be used with other sensors to increase the Boe-Bot's functionality.

Activity #1: Building and Testing the Whiskers

Parts

(2) 10 kΩ resistors
(2) 3-pin headers
(2) 3/8" 4/40 male/female standoffs
(2) ¼" 4/40 machine screws
(2) Boe-Bot bumper wires
(2) Nylon washers size #4

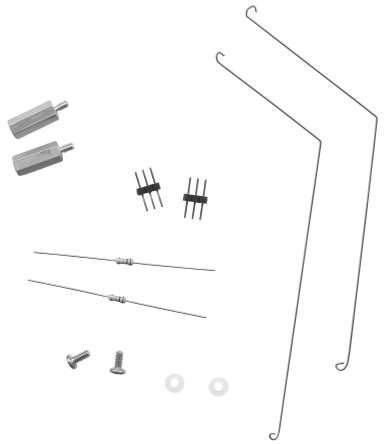


Figure 3.1 Whiskers parts

Build It!

Your #1 point Phillips screwdriver and quarter-inch combination wrench will come in handy here. Before getting started on whisker construction, take a close look at Figure 3.2. Use these pictures as a guide while constructing the mechanical part of the Whiskers kit. Figure 3.3 shows the whiskers wiring diagram. Follow it for making the necessary electrical connections.

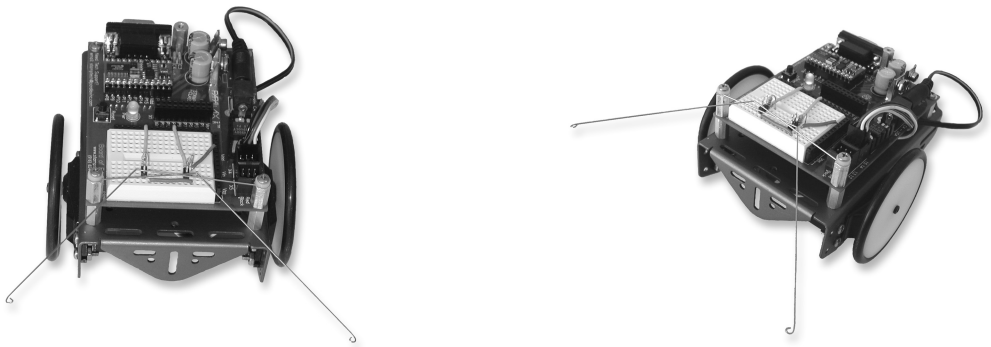


Figure 3.2: Pictures of Boe-Bot with Whiskers

- Remove the two front screws that hold your Board of Education to the front standoffs.
- Screw in the male/female standoffs included in the Whiskers kit in place of the screws that were just removed.

> ✓ **TIP** Hold the male/female standoff by the servo port while turning the standoff between the BOE and Boe-Bot chassis to tighten it. The standoff between the BOE and chassis won't turn until you loosen the screw that holds it to the chassis. Make sure to retighten it when you're done.

- Place a nylon washer on top of each standoff.

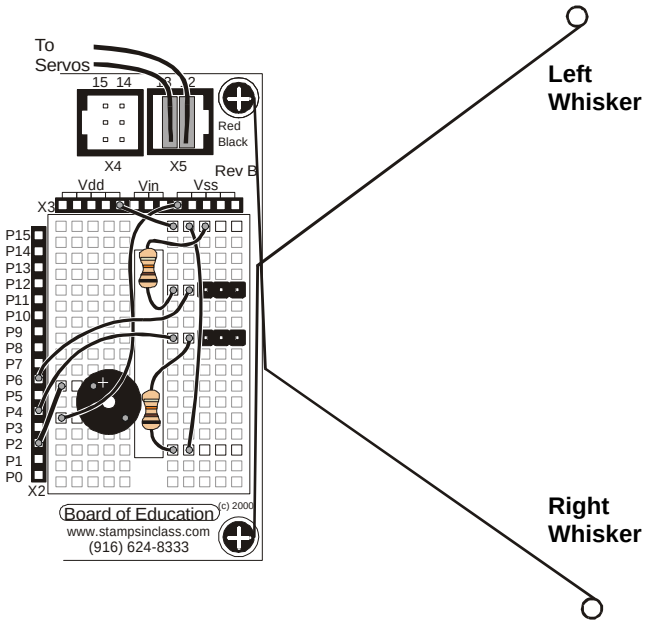


Figure 3.3: Whiskers wiring diagram.

- Thread each screw removed in the first step through the open loop of a whisker.
- Screw each screw into a standoff sandwiching the loop of the whisker between the screw head and the nylon washer. Make sure the whiskers are oriented as shown in Figure 3.2 and 3.3.

Whisker Inputs

Figure 3.4 is a schematic representation of the circuit you've just built. Each whisker is both the mechanical extension and the ground electrical connection of a normally open, single-pole, single-throw switch. 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. When a given whisker is not pressed, the voltage at the I/O pin connected to that whisker is 5 V (logic 1). When a whisker is pressed, the I/O line is shorted to ground, so the I/O line sees 0 V (logic 0).

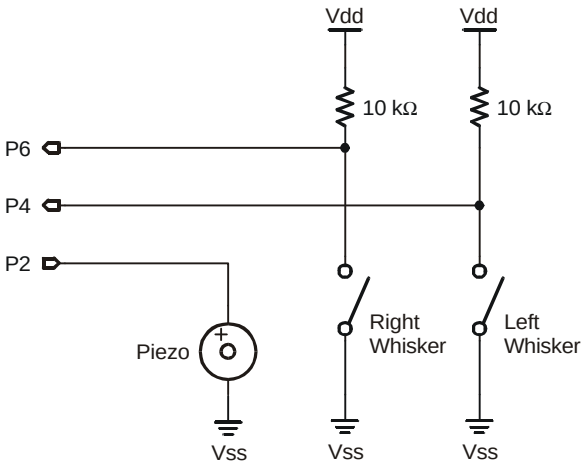


Figure 3.4: Whiskers Schematic.

Testing the Whiskers

Testing each whisker can be done with the Debug Terminal. This time, instead of displaying a printed message, the Debug Terminal is used to display the input voltage seen by the I/O pins connected to the whiskers.

Each BASIC Stamp I/O pin is connected to a memory location in RAM. You can see these memory locations by viewing the Memory Map. This is the same window used to view the BASIC Stamp's EEPROM in Chapter 2, Activity #5.

- ❑ Open the Memory Map by selecting Memory Map from the Stamp Editor's Run menu.

The top three rows in the RAM Map are memory locations connected directly to the I/O pins. The row of numbers across the top indicates the bit address for each location in a given row. The **ins**, **outs**, and **dirs** rows all refer to I/O registers.

The first row corresponds to the **ins** register, and it refers to a memory location that holds 16 bit values, each of which is either a 1 or a 0. Each bit in the **ins** register monitors the input value for an I/O pin. Bit-0 corresponds to I/O pin P0, bit-1 to P1, and so on all the way up to bit-15, which corresponds to I/O pin P15.

Each bit in the **outs** register, can be used to set the output value for a given I/O pin to either 5 V or 0 V. When a bit value in the **outs** register is a 1, the output for the corresponding I/O pin is 5 V. When the bit value is 0, the output is 0 V.

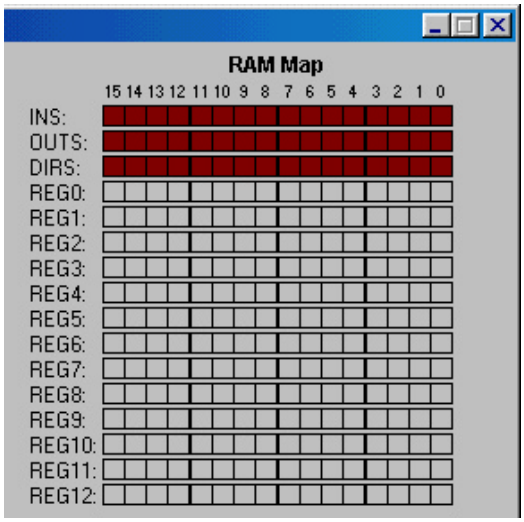


Figure 3.5: Ram Map from Memory Map Window.

Since a BASIC Stamp I/O pin can't be an input and an output at the same time, the direction of each I/O pin is set by the **dirs** register. An I/O pin is set to be either input or output, depending on the value in the **dirs** register.

Program Listing 3.1 is designed to test the whiskers to make sure they are functioning properly. It checks and displays the state of the BASIC Stamp I/O pins connected to the whiskers. 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 will cause its bit in the **ins** register to display 1 if the voltage is 5 V (whisker not pressed) and 0 if the voltage is 0 V (whisker pressed). The Debug Terminal can be used to display these values.

- Enter and run Program Listing 3.1
- This program makes use of the Debug Terminal, so leave the serial cable connected to the BOE while Program Listing 3.1 is running.

```
' Robotics! v1.5, Program Listing 3.1: Testing the Whiskers.
' {$Stamp bs2}                          ' Stamp Directive.

loop:
  debug home, "P6 = ", bin1 in6, "   P4 = ", bin1 in4
  pause 50
goto loop
```

- Note the values displayed in the Debug Terminal; it should display that both P6 and P4 are equal to 1.
- Check Figure 3.3 so you know which whisker is the "left whisker" and which whisker is the "right whisker".
- Press the right whisker so that it touches the three-pin header on the right, and note the values displayed in the Debug Terminal again. It should now read: P6 = 1 P4 = 0.
- Press the left whisker into the left three-pin header, and note the value displayed in the Debug Terminal again. This time it should read: P6 = 0 P4 = 1.
- Press both whiskers against both three-pin headers. Now it should read P6 = 0 P4 = 0.

If the whiskers passed all these tests, you're ready to move on; otherwise, check your program and circuit for errors.

How the Testing the Whiskers Program Works

The only new elements in Program Listing 3.2 are **`in6`** and **`in4`**. Instead of word or byte variables, these are bit variables. These bit variables are special because each stores the input value of a particular I/O pin. The variable **`in4`** refers to bit-4 in the **`ins`** register and **`in6`** refers to bit-6. When I/O pin P4 on the BASIC Stamp detects 0 V, the value of **`in4`** is 0. When I/O pin P4 detects 5 V, the value of **`in4`** is 1. The **`in6`** variable works the same way, except that it is monitoring P6 instead of P4.

Your Turn

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. One way of doing this would be with a pair of LED circuits and a program that turns the LEDs on and off based on the whisker inputs. Figure 3.6 shows the parts of an LED circuit along with their schematic symbols.

Extra Parts
(2) Red LEDs
(2) 470 Ω resistors

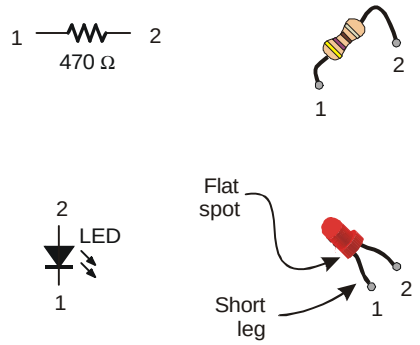


Figure 3.6 Extra parts for testing the whiskers.

FYI	LED stands for light emitting diode. The terminal labeled 1 in figure 3.6 is the LED's cathode, and the terminal labeled 2 is the LED's anode. You can usually figure out which of the LED's two wire leads is connected to the cathode because it's shorter.

✓ **TIP**	Just above where the wire leads come out of LED's plastic case, the outer rim looks round, but if you look carefully, there's a small area that's been milled flat. The lead that comes out of the plastic case closest to the milled flat spot is the cathode. If the LED's leads have been cut to the same length, look for the flat spot to figure out which lead connects to the LED's cathode.

- ❑ Construct the circuit shown in Figure 3.7.

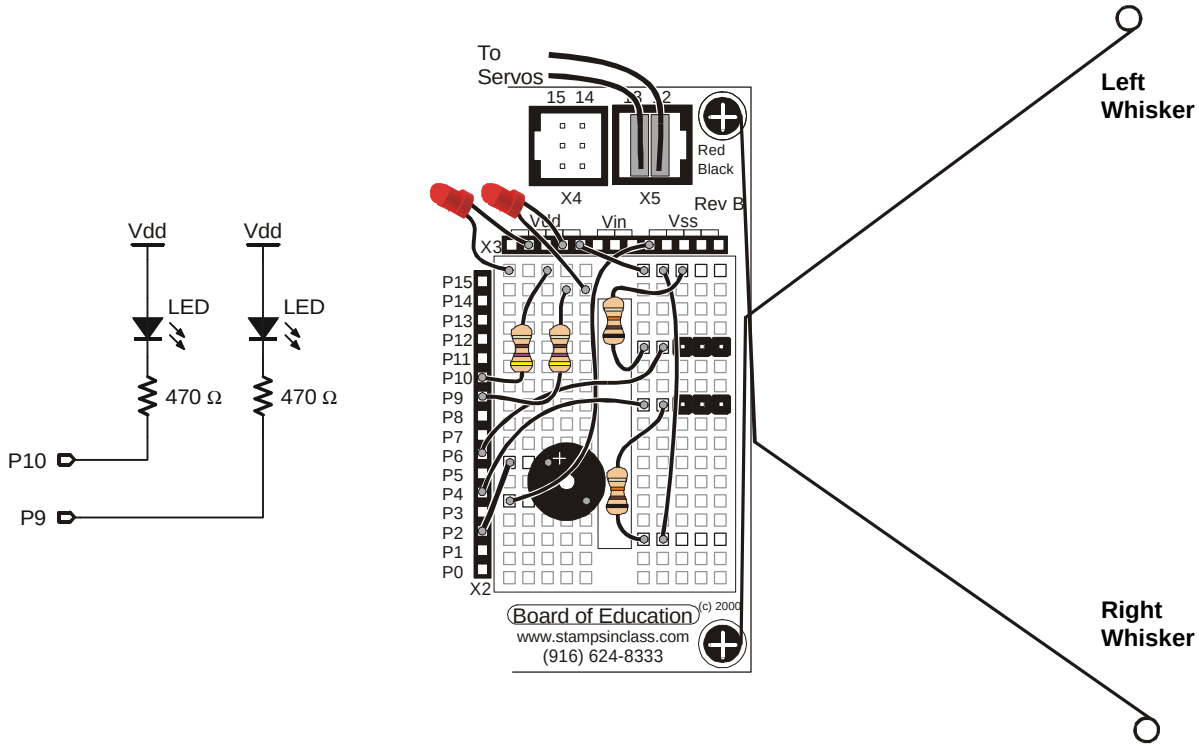


Figure 3.7 (a): add this LED circuit, (b) so the Whiskers circuit looks like this when you're done.

- Add these commands to the beginning of Program Listing 3.1:

```
output 9
output 10
```

These commands change the direction of P9 and P10 from input to output. Now instead of listening for signals, they will be ready to send signals.

- Add these two commands immediately after the **debug** command in Program Listing 3.1:

```
out9 = in4
out10 = in6
```

These statements set the output values of P9 and P10 equal to the input values at **in4** and **in6** respectively. If **in4 = 1**, **out9** is set to 1. This means that when **in4** sees 5 V, **out9** sends 5 V. If **in4** is 0, which means it detects 0 V, then **out9** will also be 0, sending 0 V.

- ❑ Run your modified version of Program Listing 3.1, and test the whiskers using the LEDs to indicate that the BASIC Stamp has detected a whisker being pressed.

Activity #2: Navigation With Whiskers

In Activity #1, the BASIC Stamp was programmed to detect whether or not a given whisker was pressed. In this activity, the BASIC stamp will be programmed to use 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 navigational routine to back up from the obstacle and go in a different direction.

Programming the Boe-Bot to Navigate Based on Whisker Inputs

Program Listing 3.2 is called a roaming program. The 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, navigational routines and subroutines developed in Chapter 2 are used to make the Boe-Bot back up and turn. Then, the Boe-Bot resumes forward motion until it bumps into another obstacle.

When a whisker is pressed, due to an obstacle, the normally open switch closes. I/O pins P6 and P4 are set to input and used to monitor the states of the whiskers. The two whiskers may be in one of four states:

(1) Both high – no objects detected
(2) Left low, right high – object detected on the left
(3) Right low, left high – object detected on the right
(4) Both low – indicates a head-on collision with a wide object such as a wall

Program Listing 3.2 shows an example of how the states of the whiskers can be used to select the appropriate Boe-Bot navigation routine. For example, state 1 means the Boe-Bot can continue forward. State 2 means that the Boe-Bot should back up, then turn right. State 3 means the Boe-Bot should back up and turn left, and state 4 would be a good time to back up and make a U-turn.

- ❑ Run Program Listing 3.2, and see how the Boe-Bot behaves when it bumps into a wall.

```
' Robotics! v1.5, Program Listing 3.2: Roaming with Whiskers.
' {$Stamp bs2}                                  ' Stamp Directive.

'----- Declarations --------------

  pulse_count  var   byte                       ' For...next loop counter.

'----- Initialization ------------

  output 2                                      ' Set P2 to output.
  freqout 2, 2000, 3000                         ' Start/restart signal.
  low 12                                        ' Set P12 and 13 to output-low.
  low 13

'----- Main Routine --------------

main:

  check_whiskers:                               ' Check each whisker.

    if in6  = 0 and in4 = 0 then u_turn         ' Backwards if both switches close.
    if in6  = 0 then right_turn                 ' Right if left switch closes.
    if in4  = 0 then left_turn                  ' Left if right switch closes.

  forward:                                      ' If no detect, one forward pulse.
    pulsout 12,500
    pulsout 13,1000
    pause 20

goto main                                       ' Check again.

'----- Navigation Routines -------

  left_turn:                                    ' Left turn routine.
    gosub backward                              ' Call Backward: before turning.
    for pulse_count = 0 to 35
      pulsout 12, 500
      pulsout 13, 500
      pause 20
    next
  goto main

  right_turn:                                   ' Right turn routine.
    gosub backward                              ' Call Backward: before turning.
    for pulse_count = 0 to 35
      pulsout 12, 1000
      pulsout 13, 1000
```

```
      pause 20
    next
  goto main

  u_turn:                                         ' U-turn routine.
    gosub backward                                ' Call Backward: before turning.
    for pulse_count = 0 to 70
      pulsout 12, 500
      pulsout 13, 500
      pause 20
    next
  goto main

'----- Navigation Subroutine -----

  backward:                                       ' Used by each navigation routine.
    for pulse_count = 0 to 70
      pulsout 12, 1000
      pulsout 13, 500
      pause 20
    next
  return
```

The mechanical design of the whiskers is by no means foolproof. Table 3.1 lists common problems you may encounter with some suggested solutions.

Table 3.1: Whisker Troubleshooting

Problem	Try This
Boe-Bot backs up too far/not far enough.	Adjust the **`for. . . next`** arguments in the program listing. They may be increased or decreased to increase or decrease how far the Boe-Bot rotates when it turns/backs up.
Boe-Bot drives up side of wall because whiskers didn't catch hold of an object.	Increase the resistance of a whisker against the surface of an object by bending whiskers in a different angle. Alternatively, try dipping the whiskers in a coating material such as rubber cement.
Dead-center object isn't detected by whiskers.	Bend whiskers inward.
Switches don't appear to work properly.	Repeat Activity #1.

How Roaming with Whiskers Works

The **if...then** statements in the **main** routine first check the whiskers for any states that require attention. If both whiskers are pressed, the **u_turn** routine is called. If just the left whisker is pressed, it contacts the post connected to P6, and the **right_turn** routine is called. If the right whisker is pressed, it contacts the post connected to P4, and the **left_turn** routine is called.

```
main:

  check_whiskers:
    if in6  = 0 and in4 = 0 then u_turn
    if in6  = 0 then right_turn
    if in4  = 0 then left_turn
```

If none of the whiskers are pressed, the default action is to execute a modified version of the **forward** routine. The **forward** routine has been modified so that it only delivers a single pulse before looping back to check the whiskers. The key to understanding how the **main** routine is structured is that it checks the whiskers between every pulse sent to the servos when the Boe-Bot is moving forward. Notice that the **pulsout** and **pause** statements in the forward routine are not nested in a **for...next** loop. Just a single pulse is sent to the servos. After the pulse, the program goes to the **main:** label, and checks the whiskers again. This process happens quickly enough that it's not even important to reduce the **pause** ***period*** in the forward routine.

```
  forward:

    pulsout 12,500
    pulsout 13,1000
    pause 20

  goto main
```

When the whiskers detect an object, the Boe-Bot is already too close to the object. To get around the object, the Boe-Bot has to back up first. This is easy to fix with a subroutine. For example, the **left_turn** routine shown below calls the **backward** subroutine using the command **gosub backward**. The great thing about subroutines is that they return program control to the line immediately following the subroutine call. So, after **backward** is done, the **for...next** loop in the **left_turn** routine begins. Another useful feature of having a **backward** subroutine is that it returns program control to the right place. If **right_turn** instead of **left_turn** calls a subroutine labeled **backward:**, program control is returned to the **right_turn** routine.

```
left_turn:
    gosub backward
    for pulse_count = 0 to 35
      pulsout 12, 500
      pulsout 13, 500
      pause 20
    next
  goto main
```

The **backward:** subroutine is just a modified version of the **backward:** routine developed in Chapter #2, Activity #2. Instead of ending with the **goto main** command, it ends with the **return** command. The **return** command works well in this situation because it returns program control to whichever of the three navigation routines called it.

```
backward:
  for pulse_count = 0 to 70
    pulsout 12, 1000
    pulsout 13, 500
    pause 20
  next
return
```

Your Turn

The **for...next** loop ***end*** arguments in the **right_turn** and **left_turn** routines can be adjusted for more or less turn, and the backward routine can have its ***end*** value adjusted to back up less for navigation in tighter spaces.

- ❑ Experiment with the **for...next** loop ***end*** argument values in navigation routines in Program Listing 3.2.

Activity #3: Looking at Multiple Inputs as Binary Numbers

Here is another way to look at States 1 through 4 discussed in Activity #2. Instead of viewing them as states, view them as 2-bit binary numbers. For those of you unfamiliar with binary counting numbers, the binary numbers 00, 01, 10, and 11 are equal to 0, 1, 2, and 3 in decimal. Table 3.2 shows how the whiskers can create these four binary numbers.

Table 3.2: Whisker States in Binary

Binary Value of state	Decimal Value of State	Branch Action Based on Value of State
0000	0	**`in6 = 0`** and **`in4 = 0`** Both whiskers detect object, execute **`u_turn`**
0001	1	**`in6 = 0`** and **`in4 = 1`** Left whisker detects object, execute **`right_turn`**
0010	2	**`in6 = 1`** and **`in4 = 0`** Right whisker detects object, execute **`left_turn`**
0011	3	**`in6 = 1`** and **`in4 = 1`** Neither whisker detects object, execute **`forward`**

Reprogramming for Roaming with Whiskers

Program Listing 3.3 does the same thing as Program Listing 3.2. However, it processes the whisker inputs by placing them in the lower two bits of a variable. This variable now stores a binary number that is then used by a branch command to call the appropriate navigation routine.

```
' Robotics! v1.5, Program Listing 3.3: Roaming with Whiskers Again.
' {$Stamp bs2}                                  ' Stamp Directive.

'----- Declarations --------------

  pulse_count  var   byte                       ' For...next loop counter.
  state        var   nib

'----- Initialization ------------

  output 2                                      ' Set P2 to output.
  freqout 2, 2000, 3000                         ' Signal program start/restart.
  low 12                                        ' Set P12 and 13 to output-low.
  low 13

'----- Main Routine --------------

main:

  check_whiskers:

    state.bit1 = in6                            ' Create binary number by saving in6
    state.bit0 = in4                            ' in state.bit1 & in4 in state.bit0.
```

```
    ' Branch to a navigation routine based on the value of the state variable.
    branch state, [u_turn, right_turn, left_turn, forward]

    forward:                                    ' Deliver a single forward pulse.
      pulsout 12,500
      pulsout 13,1000
      pause 20

    goto main

'----- Navigation Routines -------

  left_turn:                                    ' Left turn routine.
    gosub backward                              ' Call backward: before turning.
    for pulse_count = 0 to 35
      pulsout 12, 500
      pulsout 13, 500
      pause 20
    next
  goto main

  right_turn:                                   ' Right turn routine.
    gosub backward                              ' Call backward: before turning.
    for pulse_count = 0 to 35
      pulsout 12, 1000
      pulsout 13, 1000
      pause 20
    next
  goto main

  u_turn:                                       ' U-turn routine.
    gosub backward                              ' Call backward: before turning.
    for pulse_count = 0 to 70
      pulsout 12, 1000
      pulsout 13, 1000
      pause 20
    next
  goto main
```

```
'----- Navigation Subroutine -----

  backward:                                    ' Used by all navigation routines.
    for pulse_count = 0 to 70
      pulsout 12, 1000
      pulsout 13, 500
      pause 20
    next
  return
```

How Roaming with Whiskers Again Works

- Look up the **branch** command in Appendix C: PBASIC Quick Reference or in the BASIC Stamp Manual before continuing.

A nibble (**nib**) variable named **state** is added to the declarations section of Program Listing 3.3. A nibble is 4-bits, and can store a number between 0 (binary 0000) and 15 (binary 1111). The lowest two bits of the nibble are used to store the state of each whisker. Two bits can be used to count from 0 to 3 (four values).

```
declarations:
  pulse_count  var   byte
  state        var   nib
```

The **main** routine consists of two statements followed by a **branch** command. The first statement sets the binary value of bit-1 in the **state** variable equal to the binary value from **in6**. The second sets the value of bit-0 of the **state** variable equal to the value of **in4**.

```
main:

  check_whiskers:                    ' check each whisker
    state.bit1 = in6
    state.bit0 = in4

    branch state, [u_turn, right_turn, left_turn, forward]
```

Your Turn

- Insert this **debug** command just before the **branch** command in Program Listing 3.3.

```
debug home, “binary “, bin4 state, “ = decimal “, dec1 state
```

- Set the Boe-Bot on something so that when the program runs, the wheels don't touch the ground. This is so you can leave the Boe-Bot plugged into the serial cable while the program runs.
- Run the program and observe the Debug Terminal output as you test the whiskers.

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

The corner problem can be fixed by adding a counter that tracks the transitions between states 1 and 2 in Program Listing 3.3. Program Listing 3.4 below is a modified version of Program Listing 3.3 that detects and counts the number of instances of one whisker detecting an obstacle immediately after the other.

```
' Robotics! v1.5, Program Listing 3.4: Escaping Corners.
' {$Stamp bs2}                          ' Stamp Directive.

'----- Declarations -------

  pulse_count  var   byte               ' For...next loop counter.
  state        var   nib                ' Stores whisker inputs as binary numbers.
  old_state    var   nib                ' Stores state from previous pulse.
  counter      var   nib                ' Counting variable for tracking events.

'----- Initialization -----

  output 2                              ' Set P2 to output.
  freqout 2, 2000, 3000                 ' Signal program is starting/restarting.
  low 12                                ' Set P12 and 13 to output-low.
  low 13
  old_state = %0001                     ' Initialize old_state.

'----- Main Routine -------

main:

  check_whiskers:

    state.bit1 = in6                    ' Store whisker outputs in state.
    state.bit0 = in4
```

```
    ' Later in the program, a counter increments each time alternate whiskers are
    ' pressed.  The if...then statement below tests to see if the same whisker was
    ' pressed twice in a row.  If yes, then reset counter variable.

    if old_state <> state then no_reset        ' If same whisker twice then reset.
    counter = 0                                ' Reset counter.

    no_reset:                                  ' Label for if...then to skip to.

    ' If old_state exclusive-or state is not binary-0011, then skip to
    ' the continue: label.  Otherwise increment counter and decide if it's time
    ' to do a u-turn.

    if old_state ^ state <> %0011 then continue

      counter = counter +1
      old_state = state                        ' Save copy of state before update.
      if counter = 4 then u_turn

    ' Reset counter & branch based on value of state.

    continue:

      branch state, [u_turn, right_turn, left_turn, forward]

    forward:
      pulsout 12,500
      pulsout 13,1000
      pause 20

    goto main

'----- Navigation Routines -------

  left_turn:                                   ' Left turn routine.
    gosub backward                             ' Call backward: before turning.
    for pulse_count = 0 to 35
      pulsout 12, 500
      pulsout 13, 500
      pause 20
    next
  goto main

  right_turn:                                  ' Right turn routine.
    gosub backward                             ' Call backward: before turning.
    for pulse_count = 0 to 35
```

```
      pulsout 12, 1000
      pulsout 13, 1000
      pause 20
    next
  goto main

  u_turn:                                         ' U-turn routine.
    gosub backward                                ' Call backward: before turning.
    for pulse_count = 0 to 70
      pulsout 12, 500
      pulsout 13, 500
      pause 20
    next
    counter = 0
  goto main

'----- Navigation Subroutine -----

  backward:                                       ' Used by each navigation routine.
    for pulse_count = 0 to 70
      pulsout 12, 1000
      pulsout 13, 500
      pause 20
    next
  return
```

How Escaping Corners Works

Nibble variables named **old_state** and **counter** are added to the **declarations** section for keeping a running history on turns. **Old_state** tracks the previous state after the **state** variable is updated with the current whisker values. **Counter** keeps track of how many times in a row **state** transitions between one and two and back to one again.

```
declarations:
  pulse_count  var   byte
  state        var   nib
  old_state    var   nib
  counter      var   nib
```

The **check_whiskers** routine is to designed check the whiskers as before, but code also has been added that detects the "stuck in a corner" condition. This condition is indicated by four right-left-right (or left-right-left) transitions. Before updating the **state** variable with new whisker values, the statement **old_state = state** saves a copy of the previous state for comparison. Then the **state** variable is updated with the current whisker readings, just as in Program Listing 3.3.

What's

Exclusive-or

The term exclusive-or (XOR) refers to one of the operations in Boolean Algebra, which is used on binary numbers and expressions. Here's how XOR works:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

The PBASIC operator for XOR is the ^ character. When you have an expression such as variable_1 ^ variable 2, it means that each bit in variable_1 gets exclusive-orred with the corresponding bit in variable_2. In other words, bit-0 of variable_1 gets exclusive-orred with bit_0 of vriable_2. Bit-1 of variable_1 gets exclusive-orred with bit-1 of variable_2 and so on.

The NOT operator is another Boolean operator introduced in this chapter's Questions and Projects section. The BASIC Stamp Manual introduces more Boolean operators in the Binary Operators section. Boolean operators are further discussed in the section that explains the **if...then** command.

```
check_whiskers:

  old_state = state

  state.bit1 = in6
  state.bit0 = in4
```

The stuck in a corner condition rests on the assumption that alternate whiskers are pressed several consecutive times. However, if the Boe-Bot encounters an obstacle with the same whisker twice in a row, it must not be stuck in a corner. If this happens, the program should start counting from zero again. The command if **old_state <> state then no_reset** tests for the same whisker being pressed twice in a row. If a whisker has not been pressed twice in a row, the program skips to the **no_reset:** label; otherwise, it resets the counter.

```
  if old_state <> state then no_reset
  counter = 0

  no_reset:
```

Next, the program tests for whether or not an alternate whisker has been pressed by making use of the exclusive-or (^) operator. If the **old_state** exclusive-or the new **state** is not equal to binary 0011, it means that the opposite sensor was not touched. In this case, the **if...then** statement makes the program skip to the **continue:** label.

```
  if old_state ^ state <> %0011 then continue
```

On the other hand, if the statement is not true (if **old_state** XOR **state** is equal to binary 0011), then the **counter** variable is incremented. Another **if...then** tests to see if **counter** has reached 4. If it has, the **u_turn** routine is executed. If not the program simply branches to the appropriate maneuver based on the value of **state**.

```
check_counter:
 counter = counter +1
 if counter = 4 then u_turn

continue:
  branch state, [u_turn, right_turn, left_turn, forward]
```

When the count of successive whisker detections on opposite whiskers reaches 4, the **u_turn** routine is called. The **u_turn** routine has been modified so that it resets the counter any time it gets called. That way, the Boe-Bot can start counting from 0 each time it encounters a corner.

```
u_turn:
  gosub backward                  ' Call backward: before turning.
  for pulse_count = 0 to 75
    pulsout 12, 500
    pulsout 13, 500
    pause 20
  next
  counter = 0
goto main
```

Your Turn

One of the **if...then** statements in Program Listing 3.4 checks to see if counter has reached 4. Try reducing and increasing the ***condition*** argument of this **if...then** statement and note the effect. Also note if reducing the value has any effect on normal roaming. Reducing it to 0, 1, or 2 could cause interesting problems.

Summary and Applications

In this chapter, instead of navigating from a pre-programmed list, the Boe-Bot was programmed to navigate based on sensory inputs. In this case, the sensory inputs were whiskers. The BASIC Stamp was programmed to test these whisker sensors and display the test results using two different media, the Debug Terminal and LEDs.

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. Voltages of 5 and 0 V are transistor-transistor logic (TTL) levels, and the BASIC Stamp's input registers interpret these levels as "1" and "0," respectively.

PBASIC programs were developed to make the BASIC Stamp check the whiskers between each servo pulse. Based on the state of the whiskers, the programs' **`main:`** routines either made the Boe-Bot continue forward, or called navigational routines developed in the previous chapter to guide the Boe-Bot away from obstacles. As an example of artificial intelligence, a program was developed that enabled the Boe-Bot to detect when it got stuck in a corner. In this case, XOR Boolean logic expression and a counter simplified the task.

Real World Example

Automated sensors are all around you. When you go to a grocery or convenience store, sensors are often responsible for opening the door for you. Microcontrollers scan keypads in a fashion similar to the way the BASIC Stamp scans the whiskers to detect whether or not they have been pressed. The information is processed and results as an output. In the case of a door opener, the result is a servo- or motor-controlled door being opened.

Robotic machinery of many shapes and sizes also relies on a variety of tactile switches wired similarly to the whiskers. Robotic arms sometimes use these switches to detect when they've encountered the object they are programmed to pick up and place elsewhere. Factories use these switches to count objects on a production line, and also for aligning objects for industrial processes. In all these instances, the switches provide inputs that dictate some other form of programmed output. Be it a calculator, robot or a production line, the switch input is electronically monitored. Based on the state of the switches, the calculator display updates, the robot arm grabs the object, or the factory production line reacts with motors or servos to guide the product in a pre-programmed fashion.

Boe-Bot Application

This chapter introduced input-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 a lots of opportunities for the Boe-Bot to navigate in increasingly complex environments.

Questions and Projects

Questions

1. What kind of electrical connection is a whisker?

2. What are the three types of BASIC Stamp I/O registers?

3. If an I/O pin is set to output, what register does this effect?

4. 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?

5. What direction does an I/O pin have to be set to make an LED circuit function?

6. Which whisker is **`in6`** connected to? How about **`in4`**?

7. What command can you use to replace multiple **`if...then`** statements?

8. What are the three PBASIC programming elements that were used in this chapter to track events and make decisions based on those events?

Exercises

1. Assume that Program Listing 3.1 takes 1 ms to display each character and another ms to execute the loop. What is the sampling rate at which the BASIC Stamp checks each whisker?

2. In Program Listing 3.2, determine the sampling rate when the Boe-Bot rolls forward. Also determine the sampling rate while the Boe-Bot is executing maneuvers. Hint: the second portion of this exercise is a "trick question."

3. Assume that the label of the **`left_turn`** routine was changed to **`turn_left`**. Write down any necessary changes to the relevant commands in Program Listings 3.2, 3.3, and 3.4.

4. Describe the **debug** display after making the modification in the Your Turn section of Activity #3.

Projects

1. The XOR operator was introduced as a Boolean operator in this chapter. The PBASIC NOT operator is another Boolean operator denoted by the tilde character (~). You can type it on most computer keyboards by pressing Shift and the key just below the Esc key. Here's how NOT works:

   NOT 0 = 1
   NOT 1 = 0

   In the Your Turn section of Activity #1, you used LEDs to display the states of the whiskers. When you did so, the LEDs turned on when the whiskers were pressed. Use the PBASIC NOT operator (~) to make it so that each LED is on when a whisker is not pressed. The LED should then turn off when the whisker is pressed.

   Modify the program further so that the LED flashes at 2 Hz when the whisker is pressed. Hint, you will need to use a loop and **pause** statements. Try three more modifications to Program Listing 3.1, and run the program between each modification. Make it so that the LED flashes at 4 Hz, 10 Hz, and 100 Hz when the whisker is pressed.

2. Modify Program Listing 3.2 so that the Boe-Bot moves backward slowly while both whiskers are pressed. Otherwise, it stays still. Modify the program further so that the Boe-Bot rotates counterclockwise when the left whisker is pressed and clockwise then the right whisker is pressed. When the program is finished, fine tune the speed response so that it appears that you are pushing the Boe-Bot around by its whiskers.

3. Challenge: Modify Program Listing 3.4 so that the Boe-Bot travels in a circular path. Modify it so that if you tap the inside whisker, the circular path will become 5 cm. narrower in diameter. Also make it so that if you tap the whisker on the outside of the Boe-Bot's circular path, the diameter will increase by 5 cm.

   When you've got whisker control over the diameter of the Boe-Bot's circle, program the Boe-Bot to remember this diameter, even after the power is reset. The **write** command can be used to save data to EEPROM. EEPROM data is called non-volatile. Whereas the BASIC stamp's RAM is erased with each reset (volatile), the EEPROM can save the data for use the next time the BASIC Stamp gets power (non-volatile). From the PBASIC quick reference, the format for the **write** command is:

```
write address, data
```

Since we aren't using any **data** statements in this program, EEPROM address 0 can be used to record the value of the PBASIC parameter that determined the circle's diameter.

Chapter #4: Light Sensitive Navigation with Photoresistors

4

Is Your Boe-Bot a Photophile or a Photophobe?

The photoresistors in your kit can be used to make your Boe-Bot detect variations in light level. With some programming, your Boe-Bot can be transformed into a photophile (a creature attracted to light), or a photophobe (a creature that tries to avoid light).

To sense the presence and intensity of light you'll build a couple of photoresistor circuits on your Boe-Bot. A photoresistor is a light-dependent resistor (LDR) that covers the spectral sensitivity similar to that of the human eye. 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 produces high resistance, and high illumination produces very small amounts of resistance.

The specific photoresistors included in the Boe-Bot kit are EG&G Vactec (#VT935G). If you need additional photoresistors they are available from Parallax as well as from many electronic component suppliers. See Appendix A: Boe-Bot Parts Lists and Sources. The specifications of these photoresistors are shown in Figure 4.1:

Figure 4.1: EG&G Vactec Photoresistor Specifications

Photoresistor Specifications

Resistance (Ohms)					Peak Spectral Response	V_{MAX}	Response Time @ 1 fc (ms, typ.)	
10 Lux 2850K			Dark					
Min	Typ.	Max.	Min.	Sec.	nm		Rise (1-1/e)	Fall (1/e)
20K	29.0K	38K	1M	10	550	100	35	5

Illuminance is a scientific name for the measurement of incident light. The unit of measurement of illuminance is commonly the "foot-candle" in the English system and the "lux" in the metric system. While using the photoresistors we won't be concerned about lux levels, just whether or not 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. For more information about light measurement with a microcontroller, take a look at Earth Measurements Experiment #4, Light on Earth and Data Logging.

Activity #1: Building and Testing Photosensitive Eyes

Parts

Figure 4.2 shows the new parts introduced in this experiment along with their schematic symbols. Below is a list of the parts you'll need. Both parts types of parts are nonpolar, meaning that terminals 1 and 2 as shown may be swapped without affecting the circuit.

(1) Piezoelectric speaker
(2) Photoresistors
(2) 0.1 µF capacitors
(2) 0.01 µF capacitors
(2) 220 Ω resistors
(misc.) jumper wires

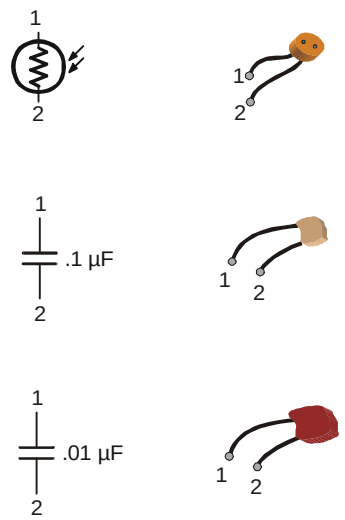
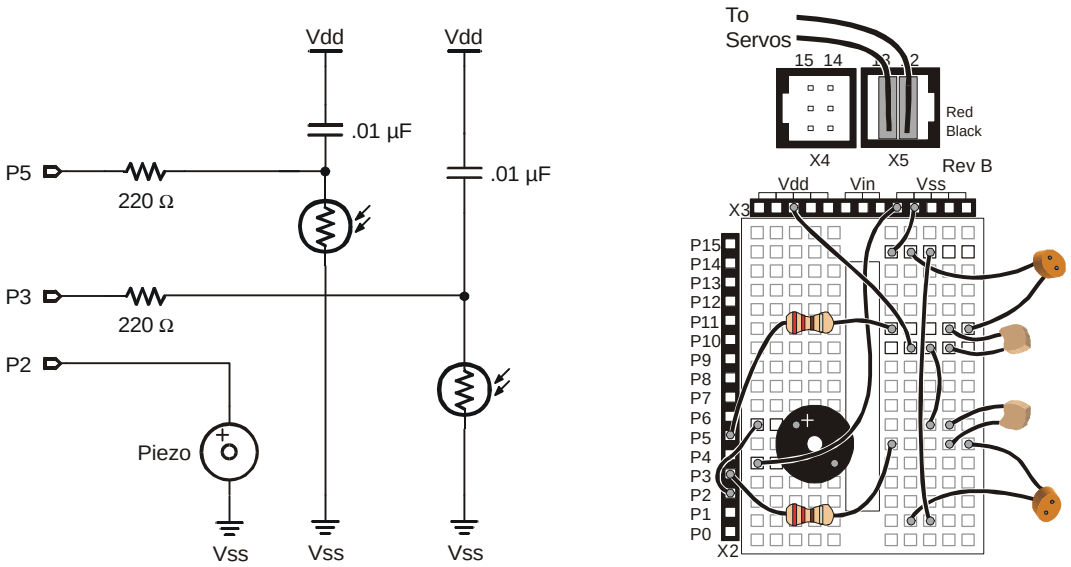


Figure 4.2: Photoresistor and capacitor circuit symbols and parts.

Build It!

Figure 4.3 shows (a) the resistor/capacitor (RC) circuit for each photoresistor and (b) a breadboard example of the circuit. A photoresistor is an analog device. Its value varies continuously as illuminance, another analog value, varies. The photoresistor's resistance is very low when it's light-sensitive surface is placed in direct sunlight. As the light level decreases, the photoresistor's resistance increases. In complete darkness, the photoresistor's value can increase to more than 1 MΩ. Even through the photoresistor is analog, its response to light is nonlinear. This means if the input source (illuminance) varies at a constant rate, the photoresistor's value does not necessarily vary at a constant rate.



(a) (b)

Figure 4.3: (a) Two photoresistor RC circuits for measurement of resistance that varies with light, and (b) breadboard example of the circuit.

> ✓ **TIP** Remember: The servo circuits are not shown in the schematics any more, but they are still shown in the breadboard diagrams. All activities from Chapter #2 onward are designed so that the servos' headers can remain plugged into servo ports 12 and 13 at all times.

Programming to Measure the Resistance

The circuit in Figure 4.3 (a) was designed for use with the PBASIC **`rctime`** command. This command can be used with an RC circuit where one value, either R or C, varies while the other remains constant. The **`rctime`** command lends itself to measuring the variable values because it takes advantage of a time varying property of RC circuits. The time it takes for the voltage on an RC circuit to change voltage depends on R × C, the RC time constant. The RC time constant is often denoted by the Greek letter Tau (τ).

For one of the RC circuits shown in Figure 4.3 (a), the first step in setting up the **`rctime`** measurement is charging the lower plate of the capacitor to 5 V. Setting the I/O pin connected to the lower capacitor plate by the 220 Ω resistor high for a few ms takes care of this. Next, the **`rctime`** command can be used to take the measurement of the time it takes the lower plate to discharge from 5 to 1.4 V. Why 1.4 V? Because that's the BASIC Stamp I/O pin's threshold voltage. When the voltage at an I/O pin set to input is above 1.4 V, the value in the input register bit connected to that I/O pin is "1." When the voltage is below 1.4 V, the value in the input register bit is "0."

The **`rctime`** command for the circuit shown in Figure 4.3 measures how long it takes for the voltage at the lower plate of the capacitor to fall from 5 to 1.4 V. This time varies according to the formula:

$$\frac{t}{R \times C} = \ln\left(\frac{V_{initial}}{V_{final}}\right)$$

$$\frac{t}{R \times 0.01 \times 10^{-6}} = \ln\left(\frac{5\ V}{1.4\ V}\right)\ \ s$$

$$t = \ln(3.57) \times R \times 0.01 \times 10^{-6}\ \ s$$

$$t = 1.27 \times 10^{-8} \times R\ \ s \qquad (4.1)$$

Equation 4.1 indicates that the time it takes the voltage at the lower plate of the capacitor in one of the Figure 4.3 (a) RC circuits to drop from 5 to 1.4 V is directly proportional to the photoresistor's resistance. Since this resistance varies with illuminance (exposure to varying levels of light), so does the time. By measuring this time, relative light exposure can be inferred.

The **`rctime`** command changes the I/O pin from output to input. As soon as the I/O pin becomes an input, the voltage at the lower plate of the capacitor starts to fall according to the time equation just discussed. The BASIC Stamp starts counting in 2 μs increments until the voltage at the capacitor's lower plate drops below 1.4 V.

✓ **TIP**	For Best Results: Eliminate direct sunlight; it's too bright for the photoresistor circuits.

- Run Program Listing 4.1. It demonstrates how to use the **rctime** command to read the photoresistors.
- This program makes use of the Debug Terminal, so leave the serial cable connected to the BOE while Program Listing 4.1 is running.


```
' Robotics! v1.5, Program Listing 4.1: Photoresistor rctime Display
' {$Stamp bs2}                                  ' Stamp Directive.

'----- Declarations --------------

  left_photo   var word                         ' For storing measured RC times of
  right_photo  var word                         ' the left & right photoresistors.

'----- Initialization ------------

  debug cls                                     ' Open and clear a Debug Terminal.

'----- Main Routine --------------

main:

  ' Measure RC time for right photoresistor.

  high 3                                        ' Set P3 to output-high.
  pause 3                                       ' Pause for 3 ms.
  rctime 3,1,right_photo                        ' Measure RC time on P3.

  ' Measure RC time for left photoresistor.

  high 5                                        ' Set P5 to output-high.
  pause 3                                       ' Pause for 3 ms.
  rctime 5,1,left_photo                         ' Measure RC time on P5.

  ' Display RC time measurements using Debug Terminal.

  debug home, "L  ", dec5 left_photo, "  R  ", dec5 right_photo

goto main
```

How Photoresistor Display Works

Two word variables, **left_photo** and **right_photo** are declared for storing the RC time values of the left and right photoresistors. The **main** routine then measures and displays the RC times for each RC circuit. The

code for reading the right RC circuit is shown below. First, the I/O pin P3 is set to output-high. Next, a 3 ms pause allows enough time for the capacitor to discharge. After 3 ms, the lower plate of the capacitor is close enough to 5 V and is ready for the **rctime** measurement. The **rctime** command measures the RC time on I/O pin P3, with a beginning state of "1" (5 V), and stores the result in the **right_photo** variable. Remember, the value stored in **right_photo** is a number. This number tells how many 2 µs increments passed before the voltage at the lower plate of the capacitor passed below the I/O pin's 1.4 V threshold.

```
high 3
pause 3
rctime 3,1,right_photo
```

Your Turn

- Try replacing one of the 0.01 µF capacitors with a 0.1 µF capacitor. Which circuit fares better in bright light, the one with the larger (0.1 µF) or the one with the smaller (0.01 µF) capacitor? What is the effect as the surroundings get darker and darker? Do you notice any symptoms that would indicate that one or the other capacitor would work better in a darker environment?

- Make sure to restore your circuit to its original state before moving on to the next activity.

Activity #2: A Light Compass

If you focus a flashlight beam in front of the Boe-Bot, the circuit and programming techniques just discussed can be used to make the Boe-Bot turn so that it's pointing at the flashlight beam. Make sure the photoresistors are pointed so that they can make a light comparison. Aside from each being pointed 45° outward from the center-line of the Boe-Bot, they also should be oriented so they are pointing 45° downward from horizontal. In other words, point the faces of the photoresistors down toward the table top. Then, use a bright flashlight to make the Boe-Bot track the direction of the light.

Programming the Boe-Bot to Point at the Light

Getting the Boe-Bot to track a light source is a matter of programming it to compare the value measured at each photoresistor. Remember that as the light gets dimmer, the photoresistor's value increases. So, if the photoresistor value on the right is larger than that of the photoresistor on the left, it means it's brighter on the left. Given this situation, the Boe-Bot should turn left. On the other hand, if the **rctime** of the photoresistor on the left is larger than that of the photoresistor on the right, the right side is brighter and the Boe-Bot should turn right.

To keep the Boe-Bot from changing directions too often, a parameter for deadband is introduced. Deadband is a range of values wherein the system makes no attempt at correction. If the numbers go above or below the deadband, then the system corrects accordingly. The most convenient way to measure for deadband is to subtract the left **rctime** from the right **rctime**, or visa versa, then take the absolute value. If this absolute value is within the deadband limits, then do nothing; if otherwise, program an appropriate adjustment.

- Enter and run Program Listing 4.2.
- Shine a bright flashlight in front of the Boe-Bot. When you move the flashlight, the Boe-Bot should rotate so that it's pointing at the flashlight beam.
- Instead of using a flashlight, use your hand to cast a shadow over one of the photoresistors. The Boe-Bot should rotate away from the shadow.

```
' Robotics! v1.5, Program Listing 4.2: Light Compass
' {$Stamp bs2}                                  ' Stamp Directive.

'----- Declarations --------------

  left_photo   var word                         ' For storing measured RC times of
  right_photo  var word                         ' the left & right photoresistors.

'----- Initialization ------------

  output 2                                      ' Set P2 to output.
  freqout 2, 2000, 3000                         ' Declare a variable for counting.
  low 12                                        ' Set P12 and 13 to output-low.
  low 13

'----- Main Routine --------------

main:

  ' Measure RC time for right photoresistor.

    high 3                                      ' Set P3 to output-high.
    pause 3                                     ' Pause for 3 ms.
    rctime 3,1,right_photo                      ' Measure RC time on P3.

  ' Measure RC time for left photoresistor.

    high 5                                      ' Set P5 to output-high.
    pause 3                                     ' Pause for 3 ms.
    rctime 5,1,left_photo                       ' Measure RC time on P5.
```

```
  ' Take the difference between right_photo and left_photo, then decide what to do.

    if abs(left_photo-right_photo) < 2 then main
    if left_photo > right_photo then right_pulse
    if left_photo < right_photo then left_pulse

'----- Navigation Routines -------

left_pulse:                                     ' Apply one pulse to left then
  pulsout 12, 500
  pulsout 13, 500
  pause 20
goto main                                       ' go back to main routine.

right_pulse:                                    ' Apply one pulse to right then

  pulsout 12, 1000
  pulsout 13, 1000
  pause 20
goto main                                       ' go back to main routine.
```

How the Light Compass Works

Program Listing 4.2 takes RC time measurements and first checks to see if the difference between the values returned by the **rctime** commands fall in the deadband using the command:

```
if abs(left_photo - right_photo) < 2 then main
```

If the difference between RC times is within the deadband, the program jumps to the **main:** label. If the measured difference in RC times is not within the deadband, two **if...then** statements decide which routine to call, **left_pulse** or **right_pulse**.

```
if left_photo > right_photo then right_pulse
if left_photo < right_photo then left_pulse
```

The **left_pulse** routine is shown below. Note that the **pulsout** commands and the **pause** command are not nested within a **for...next** loop. Instead, just one single pulse with a slightly smaller than usual **pause** is delivered, then control is returned to the **main** routine. This allows the program to check and update the photoresistor values between each servo pulse. Note that the pause value is not 20 ms. This is because each **rctime** command takes a certain amount of time, which can be subtracted from the necessary **pause**

period. The average of the combined pauses and RC times was 10 ms for the lighting conditions used in this example. Your lighting conditions are likely to be different.

```
left_pulse:
  pulsout 12, 500
  pulsout 13, 500
  pause 10
goto main
```


Your Turn

In a darker area, not only will the photoresistor values be larger, so will the difference between them. You may have to increase the deadband in low ambient light to detune the Boe-Bot to small and changing variations in light. The lower the light levels, the less you need the **pause** statements. If the Boe-Bot's performance starts to decrease, it's probably because the time between pulses has exceeded 40 ms. The first line of defense for this problem is to reduce the **pause *period*** in each subroutine to zero. The second line of defense is to check photoresistors during alternate pulses. That way, after the first pulse, the right photoresistor could be checked. Then, after the second pulse, the left photoresistor could be checked. You can try your hand at developing code that does this in this chapter's Projects section.

The deadband value is currently set to "2" in the expression:

```
if abs(left_photo-right_photo) < 2 then main
```

- Experiment with different ambient light levels and their effect on deadband by trying this experiment in lighter and darker areas. In lighter areas, the deadband value can be made smaller, even zero. In darker areas, the deadband value should be increased.

- Swap the conditions in the second and third **if...then** statement in Program Listing 4.2. Then re-run the program. Now your Boe-Bot points away from the light.

Activity #3: Follow the Light!

Simply by adding some forward motion to your Boe-Bot, you can turn it into a light-seeking robot, a photophile. An interesting experiment to try is to program the Boe-Bot to move forward and seek out light. Then, take it into a dark room with the door open to a brighter room. Assuming there are no obstacles in its way, the Boe-Bot will make its way to the door and exit the dark room.

Programming for Light Following

Programming the Boe-Bot to follow light requires that only a few modifications to Program Listing 4.2 be made. The main change is that measurements within the deadband resulted in no motion in Program Listing 4.2. In Program Listing 4.3, when the difference between RC times falls within the deadband, it results in forward motion. Let's see how it works.

```
' Robotics! v1.5, Program Listing 4.3: Light Follower
' {$Stamp bs2}                                  ' Stamp Directive.

'----- Declarations --------------

  left_photo   var word                         ' For storing measured RC times of
  right_photo  var word                         ' the left & right photoresistors.

'----- Initialization ------------

  output 2                                      ' Set P2 to output.
  freqout 2, 2000, 3000                         ' Program start/restart signal.
  low 12                                        ' Set P12 and 13 to output-low.
  low 13

'----- Main Routine --------------

main:

  ' Measure RC time for right photoresistor.

  high 3                                        ' Set P3 to output-high.
  pause 3                                       ' Pause for 3 ms.
  rctime 3,1,right_photo                        ' Measure RC time on P3.

  ' Measure RC time for left photoresistor.

    high 5                                      ' Set P5 to output-high.
    pause 3                                     ' Pause for 3 ms.
    rctime 5,1,left_photo                       ' Measure RC time on P5.

  ' Check if difference between RC times is within the deadband, 2 in this case.
  ' If yes, then forward.  If no then skip to check_dir subroutine.

  if abs(left_photo-right_photo) > 2 then check_dir

  forward_pulse:
    pulsout 12, 500
```

```
    pulsout 13, 1000
    pause 20
  goto main

  ' Jump to either right_turn or left_turn depending on which RC time is larger.

  check_dir:
    if left_photo > right_photo then right_pulse
    if left_photo < right_photo then left_pulse

'----- Navigation Routines -------

left_pulse:                                      ' Apply one pulse to left then
  pulsout 12, 500
  pulsout 13, 500
  pause 20
goto main                                        ' go back to main routine.

right_pulse:                                     ' Apply one pulse to right then

  pulsout 12, 1000
  pulsout 13, 1000
  pause 20
goto main                                        ' go back to main routine.
```

How the Light Follower Program Works

As in the previous program, the first **`if...then`** statement tests for a difference in RC time measurements within the deadband. This statement has been modified so that it skips the **`forward_pulse`** routine if the difference between RC times falls outside the deadband. On the other hand, if the difference in RC times is within the deadband, the forward pulse is executed. After the forward pulse, the program is directed back to **`main`** and the RC times are checked again.

```
if abs(left_photo-right_photo) > 2 then check_dir

  forward_pulse:
      pulsout 12, 500
      pulsout 13, 1000
      pause 20
  goto main
```

If the difference between RC times is not within the deadband, the program skips to the **`check_dir`** label. The **`if...then`** statements following the **`check_dir`** label are used to decide whether to apply a pulse to the left or a pulse to the right depending on the inequality of the **`right_photo`** and **`left_photo`** values. In this

way, the program either applies a single forward pulse or a single turn pulse each time the photoresistors are checked.

```
check_dir:
  if left_photo > right_photo then right_pulse
  if left_photo < right_photo then left_pulse
```

Your Turn

- Repeat the previous Your Turn exercise. You can now lead your Boe-Bot around with a flashlight.
- Instead of pointing the photoresistors at the surface directly in front of the Boe-Bot, point them upward and outward as shown in Figure 4.3 on Page 108. With the photoresistors adjusted this way, the Boe-Bot will roam on the floor and try to always find the brightest place.

Depending on the luminance gradient, you may have to increase the deadband to smooth out the Boe-Bot's light roaming. Alternatively, the deadband may need to be decreased to make it more responsive to seeking out the brighter areas.

Activity #4: Line Following

If the Boe-Bot can be programmed to follow a flashlight beam focused in front of it, why can't it follow a white stripe on a black background? The answer is, there's no good reason. The Boe-Bot can follow a white stripe on a black background, and it's a project in this chapter's Projects section. By the same token, the Boe-Bot should be able to follow a black stripe on a white background. Regardless of the color of the stripe, this activity is generically referred to as "line following."

The recommended width for the black stripe is about 5 cm. Either construction paper or electrical tape works fine. With some calibration along with controlled lighting conditions, the Boe-Bot is a very faithful stripe follower.

- Shadows and bright lights can be misleading, so try to keep the lighting as uniform as possible. For example, overhead fluorescent lights with no light from windows will work well.
- Also, make sure to bend the photoresistors as far over the front of the Boe-Bot as possible. In other words, readjust the photoresistors from flashlight beam following.

Programming for Line Following

By changing one value and three parameters from the previous example program, the Boe-Bot can now follow bold, black stripes on a white background. Program Listing 4.4 demonstrates this. In the comments at

the beginning of the program, data from tests performed using Program Listing 4.1 are shown. The white **rctime** readings were taken while the photoresistors were looking at a white surface in front of the Boe-Bot. The black **rctime** readings were taken while the photoresistors were looking at a black surface. The average difference between the black and white readings is 77 in this example. The average can go as low as 45, and the example should still work without a hitch. When the difference is smaller, the deadband value will have to be decreased. When the difference is larger, the deadband will have to be increased for better performance.

- Test the photoresistors using Program Listing 4.1. Use the information gathered to guess at a deadband value for stripe following.

- Adjust the deadband value to your predictions, then run your modified version of Program Listing 4.4. Try different deadband values until you find one that makes your Boe-Bot perform well when it follows the stripe.

> ✓ **TIPS**
>
> If your measurements are significantly larger than those shown in the comments section of Program Listing 4.4, you may need to increase your deadband. If your measurements are significantly smaller, decreasing the deadband will be helpful.
>
> In brightly lit rooms, decreasing the deadband value may not be enough. The 0.1 µF capacitors can be substituted for the 0.01 µF capacitors in the Boe-Bot's RC circuits. This will increase the RC times by a factor of 10. Keep this in mind when adjusting the deadband.

```
' Robotics! v1.5, Program Listing 4.4: Black Stripe Follower
' {$Stamp bs2}                                  ' Stamp Directive.

' Program Listing 4.1 readings with Boe-Bot looking at white/black surfaces.

' color             left         right
' white             58           67
' black             127          152

'----- Declarations --------------

  left_photo   var word                         ' For storing measured RC times of
  right_photo  var word                         ' the left & right photoresistors.

'----- Initialization ------------

  output 2                                      ' Set P2 to output.
  freqout 2, 2000, 3000                         ' Program start/restart signal.
```

```
  low 12                                        ' Set P12 and 13 to output-low.
  low 13

'----- Main Routine --------------

main:

  ' Measure RC time for right photoresistor.

    high 3                                      ' Set P3 to output-high.
    pause 3                                     ' Pause for 3 ms.
    rctime 3,1,right_photo                      ' Measure RC time on P3.

  ' Measure RC time for left photoresistor.

    high 5                                      ' Set P5 to output-high.
    pause 3                                     ' Pause for 3 ms.
    rctime 5,1,left_photo                       ' Measure RC time on P5.

  ' Check if difference between RC times is within the deadband, 7 in this case.
  ' If yes, then forward.  If no then skip to check_dir subroutine.

  ' IMPORTANT: The deadband value (currently 7) should be made smaller in
  ' brighter rooms and larger in darker rooms.  Otherwise, the Boe-Bot will not
  ' detect the black stripe.  It will take some fine tuning to optimize the
  ' Boe-Bot’s ability to follow stripes.

    if abs(left_photo-right_photo) > 7 then check_dir

    forward_pulse:
      pulsout 12, 500
      pulsout 13, 1000
    pause 20
    goto main

  ' Jump to either right_turn or left_turn depending on which RC time is larger.

    check_dir:
      if left_photo < right_photo then right_pulse
      if left_photo > right_photo then left_pulse

'----- Navigation Routines -------

left_pulse:                                     ' Apply one pulse to left then
  pulsout 12, 500
  pulsout 13, 500
  pause 20
```

```
goto main                                        ' go back to main routine.

right_pulse:                                     ' Apply one pulse to right then

  pulsout 12, 1000
  pulsout 13, 1000
  pause 20
goto main                                        ' go back to main routine.
```


How the Black Stripe Follower Program Works

The three lines below are the only ones changed between Program Listing 4.3 and Program Listing 4.4. The deadband was increased from "2" to "7," and the inequality signs (< and >) of the lower two **`if...then`** statements were swapped. As noted earlier, that's all it takes to change a robotic light seeker (photophile) into a light avoider (photophobe). Light avoidance is key to this program.

```
        if abs(left_photo-right_photo) > 7 then check_dir
                  .
                  .
                  .
          if left_photo < right_photo then right_turn
          if left_photo > right_photo then left_turn
```

Your Turn

Try a black stripe with a 45° turn in the middle of it.

Try a black strip with a 90° turn in it, and see if you can pick a deadband that will navigate it.

Remember, you may need to adjust your deadband to succeed in these maneuvers.

For either or both of the maneuvers above, find the upper and lower limits of deadband values with which the Boe-Bot still can successfully navigate.

Summary and Applications

This chapter focused on measuring the difference in light intensity and using it as a guide for the Boe-Bot. The PBASIC command **`rctime`** was used in conjunction with an RC circuit to measure each photoresistor. The exact resistance value of each photoresistor was disregarded in favor of the relative difference between the two values. This difference is a simple subtraction problem, but it can be used to gage which direction has a higher illuminance.

Real World Example

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.

Deadband is often a problem in navigation control systems. In terms of tracking and controlling machinery, deadband can result from the uncertainty in measurements due to mechanical connections. The result is that deadband is the area you don't know about and try to develop creative ways of dealing with it. On the other hand, deadband is also the way a thermostat works. In the context of maintaining temperature, differential gap control uses a built-in deadband region where no correction is made to the temperature.

Boe-Bot Application

As you can see, the Boe-Bot can do an interesting variety of tricks with a pair of photoresistors as its guide. It can point at light, move itself from a dark place into a light place, follow a guiding flashlight beam and follow a black stripe with turns in it on a white piece of paper. That's not bad for some inexpensive photoresistors, capacitors and resistors.

Questions and Projects

Questions

1. Name and describe the element in the photoresistors that changes resistance in response to illuminance.

2. What does the BASIC Stamp measure to infer the resistance in an RC circuit? What value must remain fixed in an RC circuit to infer a variable resistance? Why?

3. What are the increments of the **`rctime`** measurement?

4. When the value of a photoresistor increases, what does that indicate?

5. How does the program for a light following Boe-Bot differ from that of a dark following Boe-Bot?

6. What role does deadband play in the Boe-Bot's tendency to move forward? What role does it play in the Boe-Bot's tendency to change direction?

Exercises

1. If you have a 10 μF capacitor and your **`rctime`** value is 150, what is the resistance of the photoresistor? Hint: Use equation 4.1.

2. Take a look at the **`if...then`** statement setting deadband in Program Listing 4.2. If you want a deadband that spans differences in RC time measurements of –6 to +6, what argument will you have to use in what statement?

3. Re-derive Equation 4.1 using a 0.1 μF capacitor. What kinds of problems arise if the 0.1 μF capacitors replace the 0.01 μF capacitors? What effect does the increased RC value have on the measurement time? What effect does the measurement time have on servo performance?

Chapter #5: Object Detection Using Infrared

Using Infrared Headlights to See the Road

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. With a few inexpensive and widely available parts, the BASIC Stamp can also use an infrared LED and detector to detect objects to the front and side of your traveling Boe-Bot.

What's

Infrared

Infra means below, so Infra-red is light (or electromagnetic radiation) that has lower frequency, or longer wavelength than red light. 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.

Color	Approximate Wavelength
Violet	400 nm
Blue	470
Green	565
Yellow	590
Orange	630
Red	780
Near infra-red	800-1000
Infra-red	1000-2000
Far infra-red	2000-10,000nm

Detecting obstacles doesn't require anything as sophisticated as machine vision. A much simpler system will suffice. 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 easily available and inexpensive.

The Boe-Bot infrared object detection scheme has a variety of uses. The Boe-Bot can use infrared to detect objects without bumping into them. As with the photoresistors, infrared can be used to detect the difference between black and white for line following. Infrared can also be used to determine the distance of an object from the Boe-Bot. The Boe-Bot can use this information to follow objects at a fixed distance, or detect and avoid high ledges.

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 5.1. 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 to the BASIC Stamp 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 input.

The IR detectors have built-in optical filters that allow very little light except the 980 nm. infrared that we want to detect onto 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 flashed on and off at 38,500 times per second. This prevents interference from common IR interference sources such as sunlight and indoor lighting. Sunlight is DC interference (0 Hz), and house lighting tends to flash on and off at either 100 or 120 Hz, depending on the main power source in the country where you reside. Since 120 Hz is way outside the electronic filter's 38.5 kHz band pass frequency, it is, for all practical purposes, completely ignored by the IR detectors.

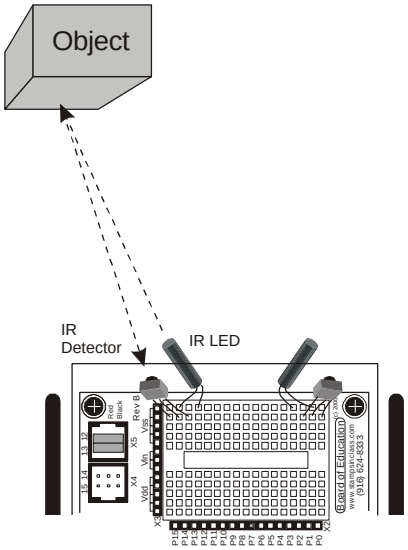


Figure 5.1: Object detection with IR Headlights.

The Freqout Trick

Since the IR detectors only see IR signals in the neighborhood of 38.5 kHz, the IR LEDs have to be flashed on and off at that frequency. A 555 timer can be used for this purpose, but the 555 timer circuit is more complex and less functional than the circuit we will use in this and the next chapter. For example, the method of IR detection introduced here can be used for distance detection; whereas, the 555 timer would need additional hardware to do distance detection.

A pair of Boe-Bot enthusiasts found an interesting trick that made the 555 timer scheme unnecessary. This scheme uses the **`freqout`** command without the RC filter that's normally used to smooth the signal into a sine-wave. Even though the highest frequency **`freqout`** is designed to transmit is 32768 Hz, the unfiltered **`freqout`** output contains a harmonic with useful properties for a 38.5 kHz IR detector. More useful still is the fact that you can use a command such as **`freqout pin, period, 38500`** to send a 38.5 kHz harmonic that the IR detector will detect.

Figure 5.2 shows (a) the signal sent by the command **`freqout pin, period, 27036`**. Tuned electronic receivers, such as the IR detectors we'll be using, can detect components of this signal that are called harmonics. The **`freqout`** signal's two dominant low frequency harmonics are shown in Figures 5.2 (b) and (c). Figure 5.2 (b) shows the fundamental harmonic, and Figure 5.2 (c) shows the third harmonic. These harmonics are actually components of the unfiltered **`freqout`** pulses shown in Figure 5.2 (a). The third harmonic shown in Figure 5.2 (c) can be controlled directly by entering commands such as **`freqout pin, period, 38500`** (instead of **`27036`**) for 38.5 kHz, or **`freqout pin, period, 40000`** for 40 kHz, etc.

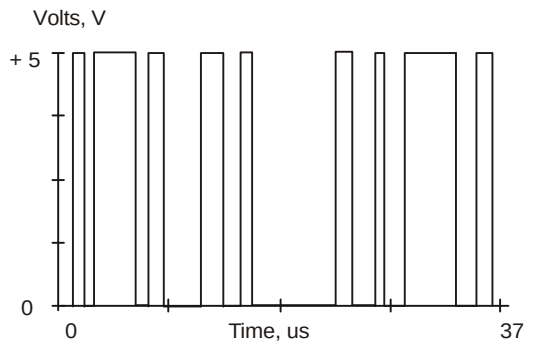
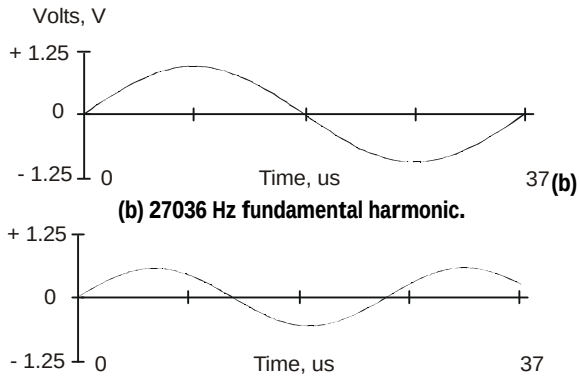


Figure 5.2: (a) Unfiltered freqout pulses sent by `freqout pin, period, 27036`



(b) 27036 Hz fundamental harmonic.

(c) 38500 Hz third harmonic.

Even though the freqout trick works, there is an additional problem. The BASIC Stamp does not multitask. The reason this is a problem is because the IR detector only sends the low signal indicating that it has detected an object while it is receiving the 38.5 kHz IR. Otherwise, it sends a high signal. Fortunately, it takes the detector long enough to rebound from its low output state that the BASIC Stamp can capture the value. The reason that the detector's output takes so long to rebound is related to its tendency toward slower responses when it receives a signal with unequal high and low times, of which the signal in Figure 5.2 (a) has many.

Activity #1: Building and Testing the New IR Transmitter/Detector

Parts

(1) Piezoelectric speaker
(2) Shrink wrapped IR LEDs
(2) IR detectors
(2) 220 Ω resistors
(misc) wires

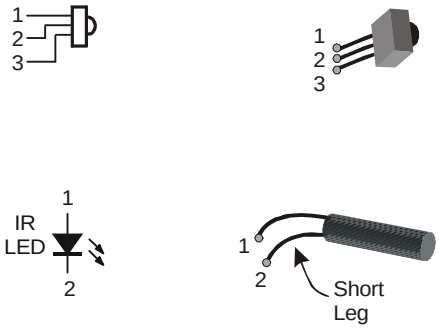
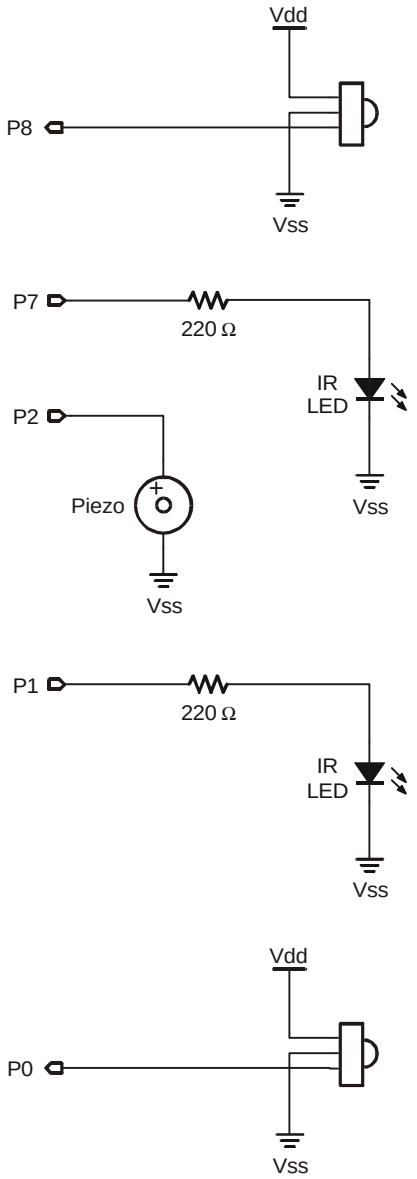


Figure 5.3: IR detector schematic symbol and part on top row and IR LED schematic symbol and part on bottom row.

Build It!



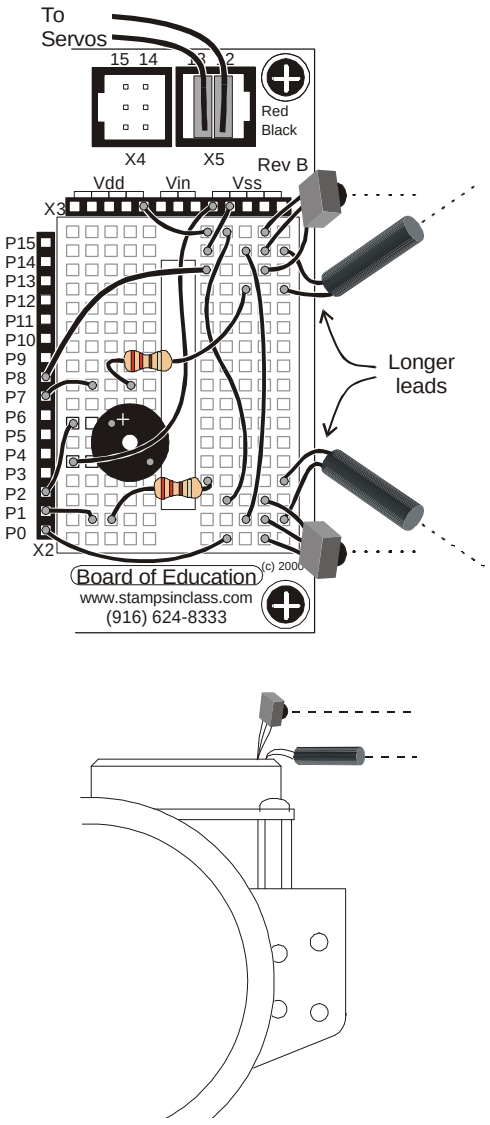


Figure 5.4: IR headlights (a) Schematic **(b) wiring diagram.**

One IR pair (IR LED and detector) is mounted on each corner of the breadboard. Figure 5.4 shows the IR headlights circuit as both a schematic and wiring diagram. Build the circuit as shown.

Testing the IR Pairs

The key to making each IR pair work is to send 1 ms of unfiltered 38.5 kHz **freqout** harmonic followed immediately by testing the signal sent by the IR detector and saving its output value. The IR detector's normal output state when it sees no IR signal is high. When the IR detector sees the 38500 Hz harmonic sent by the IR LED, it's output will drop from high to low. Of course, if the IR does not reflect off an object, the IR detector's output simply stays high. Program Listing 5.1 shows an example of this method of reading the detectors.

- Enter and run Program Listing 5.1.
- This program makes use of the Debug Terminal, so leave the serial cable connected to the BOE while Program Listing 5.1 is running.

```
' Robotics! v1.5, Program Listing 5.1: IR Pairs Display.
' {$Stamp bs2}                                  ' Stamp Directive.

'----- Declarations --------------

  left_IR_det   var  bit                        ' Two bit variables for saving IR
  right_IR_det  var  bit                        ' detector output values.

'----- Initialization ------------

  output 2                                      ' Set all I/O lines sending freqout
  output 7                                      ' signals to function as outputs
  output 1

'----- Main Routine --------------

main:
                                                ' Detect object on the left.
  freqout 7, 1, 38500                           ' Send freqout signal - left IRLED.
  left_IR_det = in8                             ' Store IR detector output in RAM.
                                                ' Detect object on the right.
  freqout 1, 1, 38500                           ' Repeat for the right IR pair.
  right_IR_det = in0

  debug home, "Left= ", bin1 left_IR_det
  pause 20
```

```
  debug "  Right= ", bin1 right_IR_det
  pause 20

goto main
```

- While program Listing 5.1 is running, point the IR detectors so nothing nearby could possibly reflect infrared back at the detectors. The best way to do this is to point the Boe-Bot up at the ceiling. The Debug output should display both left and right values as equal to "1."

- By placing your hand in front of an IR pair, it should cause the Debug Terminal display for that detector to change from "1" to "0." Removing your hand should cause the output for that detector to return to a "1" state. This should work for each individual detector, and you also should be able to place your hand in front of both detectors and make both their outputs change from "1" to "0."

- If the IR Pairs passed all these tests, you're ready to move on; otherwise, check your program and circuit for errors.

How the IR Pairs Display Program Works

Two bit variables are declared to store the value of each IR detector output. The first **`freqout`** command in the **`main`** routine is different. The command **`freqout 7, 1, 38500`** sends the on-off pattern shown in Figure 5.2 via left IR LED circuit by causing it to flash on and off rapidly. The harmonic contained in this signal either bounces off an object, or not. If it bounces off an object and is seen by the IR detector, the IR detector sends a low signal to IO pin P8. Otherwise, the IR detector sends a high signal to P8. So long as the next command after the **`freqout`** command is the one testing the state of the IR detector's output, it can be saved as a variable value in RAM. The statement **`left_IR_det = in8`** checks P8, and saves the value ("1" for high or "0" for low) in the **`left_IR_det`** bit variable. This process is repeated for the other IR pair, and the IR detector's output is saved in the **`right_IR_det`** variable. The **`debug`** command then displays the values in the debug window.

Your Turn

- Experiment with detuning your IR pairs by using frequencies above 38.5 kHz. For example, try 39.0, 39.5, 40.0, 40.5 and 41 kHz. Note the maximum distance that each will detect by bringing an object progressively closer to the IR pairs and noting what distance began to cause the IR detector output to switch from "1" to "0."

Activity #2: 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, Program Listing 3.2: Roaming with Whiskers is modified so that it works with the IR detectors.

Converting the Whiskers Program For IR Object Detection/Avoidance

Roaming with Whiskers can be modified so that each IR LED circuit receives the **`freqout`** signal. Immediately after sending the freqout signal the output state of the IR detector in the pair needs to be checked and recorded. After the information is recorded, it can be compared using the same **`if...then`** statements and navigation routines used in the original whiskers navigation program (Program Listing 3.2).

```
' Robotics! v1.5, Program Listing 5.2: Roaming with Whiskers Adjusted for IR Pairs.
' {$Stamp bs2}                                  ' Stamp Directive.

'----- Declarations --------------

  pulse_count  var   byte                       ' For...next loop counter.
  left_IR_det   var  bit                        ' Two bit variables for saving IR
  right_IR_det  var  bit                        ' detector output values.

'----- Initialization ------------

  output 2                                      ' Set all I/O lines sending freqout
  output 7                                      ' signals to function as outputs
  output 1
  freqout 2, 2000, 3000                         ' Program start/restart signal.
  low 12                                        ' Set P12 and 13 to output-low.
  low 13

'----- Main Routine --------------

main:
                                                ' Detect object on the left.
  freqout 7, 1, 38500                           ' Send freqout signal - left IRLED.
  left_IR_det = in8                             ' Store IR detector output in RAM.
                                                ' Detect object on the right.
  freqout 1, 1, 38500                           ' Repeat for the right IR pair.
  right_IR_det = in0

  ' With the exception that values stored in RAM are used instead of
  ' input register values, the decision making process is the same as
  ' the one used in Program Listing 3.2.
```

```
  if left_IR_det  = 0 and right_IR_det = 0 then u_turn
  if left_IR_det  = 0 then right_turn
  if right_IR_det  = 0 then left_turn

  ' The commands from this point onward are identical to
  ' Program Listing 3.2: Roaming with Whiskers.

  forward:                                      ' If no detect, one forward pulse.
    pulsout 12,500
    pulsout 13,1000
    pause 20

goto main                                       ' Check again.

'----- Navigation Routines -------

  left_turn:                                    ' Left turn routine.
    gosub backward                              ' Call Backward: before turning.
    for pulse_count = 0 to 35
      pulsout 12, 500
      pulsout 13, 500
      pause 20
    next
  goto main

  right_turn:                                   ' Right turn routine.
    gosub backward                              ' Call Backward: before turning.
    for pulse_count = 0 to 35
      pulsout 12, 1000
      pulsout 13, 1000
      pause 20
    next
  goto main

  u_turn:                                       ' U-turn routine.
    gosub backward                              ' Call Backward: before turning.
    for pulse_count = 0 to 75
      pulsout 12, 1000
      pulsout 13, 1000
      pause 20
    next
  goto main

'----- Navigation Subroutine -----

  backward:                                     ' Used by each navigation routine.
```

```
    for pulse_count = 0 to 75
      pulsout 12, 1000
      pulsout 13, 500
      pause 20
    next
  return
```


How the Roaming with Whiskers Adjusted for IR Pairs Program Works

Two bit variables, **left_IR_det** and **right_IR_det**, are added for capturing and holding the IR detectors' output states.

```
declarations:
  pulse_count  var   byte
  left_IR_det   var  bit
  right_IR_det  var  bit
```

The **main** routine has four additional commands, two for checking the output of each IR detector. Each **freqout** command sends a 1 ms unfiltered freqout signal to the IR LED circuit in the pair. The value at the input connected to the IR detector's output is saved as a bit variable in the BASIC Stamp's RAM. For example, the command **freqout 7, 1, 38500** is followed immediately by the statement **left_IR_det = in8**. This command sets the value of **left_IR_det** equal to the input at P8, the I/O pin connected to the left IR detector's output.

```
main:

  check_IR_pairs:

    freqout 7, 1, 38500
    left_IR_det = in8

    freqout 1, 1, 38500
    right_IR_det = in0
```

The saved bit values for each IR detector output can be used in the exact same way the whiskers navigation program used them. With one exception, the navigation routines that are executed according to **if...then** statements are identical to those originally used in Program Listing 3.2: Roaming with Whiskers. The **if...then** statements themselves are changed to accommodate the need to capture and store the output from each IR detector; whereas, the **if...then** statements in the whiskers program used the input values directly.

```
    if left_IR_det  = 0 and right_IR_det = 0 then u_turn
```

```
          if left_IR_det  = 0 then right_pulse
          if right_IR_det  = 0 then left_pulse
```

Your Turn

As with Program Listing 3.2, you can fine tune the end arguments in the **`for...next`** loops to fine tune the Boe-Bot's turning and backing up behaviors.

Activity #3: Navigating by the Numbers in Real-Time

In Program Listing 5.2, the Boe-Bot checked between each **`forward`** pulse to see if it was still okay to move forward. When the Boe-Bot performed maneuvers, they were essentially pre-recorded motions. Another approach to IR navigation is to check the sensors, apply a single pulse based on the sensor input, then check the sensors again. The Boe-Bot behaves very differently using this technique.

Real-Time IR Navigation

Program Listing 5.3 checks the IR pairs and delivers one of four different pulses based on the sensors. Each of the navigational routines is just a single pulse in either the **`forward`**, **`left_turn`**, **`right_turn`** or **`backward`** directions. After the pulse is applied, the sensors are checked again, then another pulse is applied, etc. This program also makes use of some programming techniques you will find very useful.

```
' Robotics! v1.5, Program Listing 5.3:  IR Roaming by Numbers in Real Time
' {$Stamp bs2}                                  ' Stamp Directive.

'----- Declarations --------------
                                                ' The lower 2 bits of the
  sensors var nib                               ' sensors variable are used to store
                                                ' IR detector values.
'----- Initialization ------------

  output 2                                      ' Set all I/O lines sending freqout
  output 7                                      ' signals to function as outputs
  output 1
  freqout 2, 2000, 3000                         ' Program start/restart signal.
  low 12                                        ' Set P12 and 13 to output-low.
  low 13

'----- Main Routine --------------

main:
                                                ' Detect object on the left.
   freqout 7,1,38500                            ' Send freqout signal - left IRLED.
```

```
   sensors.bit0 = in8                          ' Store IR detector output in RAM.
                                               ' Detect object on the right.
   freqout 1,1,38500                           ' Repeat for the right IR pair.
   sensors.bit1 = in0

   pause 18                                    ' 18 ms pause(2 ms lost on freqout).

   ' By loading the IR detector output values into the lower 2 bits of the sensors
   ' variable, a number btwn 0 and 3 that the branch command can use is generated.

   branch sensors,[backward,left_turn,right_turn,forward]

'----- Navigation Routines -------

      forward:     pulsout 13,1000: pulsout 12,500: goto main
      left_turn:   pulsout 13,500: pulsout 12,500: goto main
      right_turn:  pulsout 13,1000: pulsout 12,1000: goto main
      backward:    pulsout 13,500: pulsout 12,1000: goto main
```

5

How IR Roaming by Numbers in Real-Time Works

- Look up the **branch** command in Appendix C: PBASIC Quick Reference or in the BASIC Stamp Manual.

This Program listing declares the **sensors** variable, which is one nibble of RAM. Of the four bits in the sensors variable, only the lowest two bits are used. Bit-0 is used to store the left detector's output, and bit-1 is used to store the right detector's output.

```
declarations:
  sensors var nib
```

I/O pins P7, P1, and P2 are declared outputs. P2 is declared an output so that freqout can send signals to the speaker. P7 and P1 are declared outputs because these lines drive the left and right IR LED circuits.

```
initialization:
  output 7
  output 1
  output 2
  freqout 2,2000,3000
```

The main routine starts with the **freqout** commands used to send the IR signals, but the commands following each freqout command are slightly different from those used in the previous program. Instead of saving the bit value at the input pin to a bit variable, each bit value is stored as a bit in the **sensors** variable. Bit-0 of

sensors is set to the binary value of **in8**, and bit-1 of the sensors variable is set to the binary value of **in0**. After setting the values of the lower two bits of the sensors variable, it will have a decimal value between "0" and "3." The **branch** command uses these numbers to determine to which label it sends the program.

```
main:

   freqout 7,1,38500
   sensors.bit0 = in8

   freqout 1,1,38500
   sensors.bit1 = in0

   pause 18

   branch sensors,[backward,left_turn,right_turn,forward]
```

The four possible binary numbers that result are shown in Table 5.1. Also shown is the branch action that occurs based on the value of the state argument.

Table 5.1: IR Detector States as Binary Numbers

Binary Value of state	Decimal Value of State	What the Value Indicates, Branch Action Based on State
0000	0	**in8 = 0** and **in0 = 0**, Both IR detectors detect object, pulse servos **backward**.
0001	1	**in8 = 0** and **in0 = 1**, Left IR detector detects object, pulse **right_turn**
0010	2	**in8 = 1** and **in0 = 0**, Right IR detector detects object, pulse for **left_turn**
0011	3	**in8 = 1** and **in0 = 1**, Neither IR detector detects object, pulse **forward**.

Depending on the value of the **sensors** variable, the **branch** command sends the program to one of four routines: **forward**, **left_turn**, **right_turn**, **or backward**. Whichever routine the program ends up in gives the servos a single pulse in the appropriate direction, after which, the routine sends the program back to the **main** routine for another check of the sensors.

```
routines:

      forward:       pulsout 13,1000: pulsout 12,500: goto main
```

```
left_turn:     pulsout 13,500: pulsout 12,500: goto main
right_turn:    pulsout 13,1000: pulsout 12,1000: goto main
backward:      pulsout 13,500: pulsout 12,1000: goto main
```

> ✓ **TIP** Each routine is a label followed by three commands, all on the same line. When you put more than one command on the same line, each must be separated by a colon. If a label appears on a line with more than one PBASIC command, it must be the first item on that line.

Your Turn

You can rearrange the address labels in the branch command so that the Boe-Bot does different things in response to obstacles. One interesting activity is to try replacing the **`backward`** address with the **`forward`** address. There will be two instances of **`forward`** in the **`branch`** address list, but this is not a problem. Also, swap the **`left_turn`** and **`right_turn`** addresses.

- ❑ Try making the changes just discussed.
- ❑ Run the modified version of Program Listing 5.3, and have the Boe-Bot follow your hand as you lead it places.

If you stop your hand, the Boe-Bot will run into it. Because of this, one Boe-Bot cannot be programmed to follow another without some way of distance detection. If the one in front stops, the one in back will crash into it. This problem will be fixed as an example in the next chapter.

Summary and Applications

This chapter covered a unique technique for infrared object detection. By shining infrared into the Boe-Bot's path and looking for its reflection, object detection can be accomplished. Infrared LED circuits are used to send a 38.5 kHz signal by using a unique and little-known property of **`freqout`**. This property allows you to control a harmonic of the **`freqout`** PWM wave-shaping signal via IR LED circuits.

PBASIC programming techniques also were covered for minimizing the time spent on capturing the IR detector's output signal. The detector's output signal has a delay that makes it possible for the BASIC Stamp to read the IR detector's output even after the **`freqout`** signal no longer is transmitting.

Navigation techniques for checking sensors between each servo pulse and binary processing of the sensor outputs also was introduced. Used together, these two techniques allow for very responsive Boe-Bot performance.

Real World Example

Infrared is one of the more popular amenities on electronic products. TV remotes, palmtop computers, and fancy calculators all use infrared for communication. A variety of communication schemes exist for transmitting data. A TV remote control, for example, sends a high signal by flashing its IR transmitter at 38.5 kHz. A low signal is no IR. The detectors in some TVs, VCRs, etc. are identical to the receiver used in the Boe-Bot.

The detection scheme in the automatic door openers common in convenience and grocery stores relies on the same theory of operation for object detection used by the Boe-Bot. Whenever you trigger one of these door openers, it's because you walked into and broke the IR beam being reflected back at the receiver. Infrared detectors also are mounted on many different conveyer belts. Factories use them to count products as they fly by, and grocery stores use them to detect when the groceries have reached the end of the conveyer belt. In grocery stores, these belts automatically move the groceries forward so the checker can reach them. To prevent the conveyer belt from piling groceries on the scanner, an IR detector is mounted at the end of the conveyer belt. When the Twix candy bar interrupts the IR beam shining across the conveyer belt, the IR detector's output changes. When this change is detected by a microcontroller, it stops the motor that runs the conveyer belt.

Boe-Bot Application

The unique thing about IR detectors is that they allow the Boe-Bot to detect objects without actually touching them. In maze competitions where you lose points by touching the walls, this is a real plus.

Questions and Projects

Questions

1. What does infrared mean? How does infrared differ from near infrared?
2. What are the two kinds of filters built into the IR detectors in the Robotics! Kit? What does each do?
3. Describe what each of the two IR detector outputs mean.
4. Why is it important to check and save the output state of the IR detector immediately after sending the 38.5 kHz signal?
5. What happens if you send a 39.5 kHz signal instead of a 38.5 kHz signal?
6. How does Program Listing 5.2 resemble Program Listing 3.2? How do the two programs differ?
7. What values would you expect to see the `sensors` variable in Program Listing 5.3 storing?
8. What character is required to separate more than one PBASIC command on the same line? What convention must always be followed with program labels when multiple commands are put on the same line?

Exercises

1. If you wanted to send a 35 kHz infrared harmonic out the Boe-Bot's IR LED circuit, what command would you use?
2. Modify Program Listing 5.2 so that the right IR pair is checked before the left pair.
3. Modify Program Listing 5.2 so that it makes the Boe-Bot follow objects instead of avoiding them. Describe the problems you encounter, if any.

Projects

1. There are two ways to make the IR pair circuit less sensitive to objects. In other words, the circuit can be made near-sighted, so that it only sees objects that are closer. The first involves a physical change: the replacement of the 220 Ω resistors with 470 Ω resistors. The second involves detuning by sending a harmonic that's not at the IR detector's center frequency of 38.5 kHz.

   - ❑ Test for the maximum detection distance for each IR pair by holding an object in front of the pair while Program Listing 5.1 is running. As you pull the object slowly away from the IR pair, make a note of the distance when the debug window started to flicker between "0" and "1."

   - ❑ Replace both 220 Ω resistors with 470 Ω resistors.

   - ❑ Repeat the first two steps and note any change in the maximum detection distance for each IR pair.

   - ❑ Remove the 470 Ω resistors and put the 220 Ω resistors back in.

   - ❑ Place the object used for testing detection at the maximum detection distances determined using the 470 Ω resistor for the first IR pair.

   - ❑ Modify the **`freqout`** ***`frequency`*** arguments by iteratively adding 25 at a time to the **`freqout`** command's ***`frequency`*** argument. After each time the frequency argument is changed, re-run Program Listing 5.1. When you start to observe the flickering 0/1 in the Debug Terminal display, it indicates the object is falling out of range.

2. One of the shortcomings of IR object detection is that the Boe-Bot's IR detectors do not detect black. That's because black absorbs IR instead of reflecting it. The Boe-Bot tends to run into black objects when roaming with IR because it doesn't see them. Add whiskers to your breadboard and modify Program Listing 5.2 so that it checks the whisker inputs before checking the IR detectors. In this way the Boe-Bot can check to see if it has run into a black object.

Remember: Wrap the portions of the whiskers with electrical tape that could come into contact with circuits other than the whisker contact posts.

Chapter #6: Determining Distance Using Frequency Sweep

What's a Frequency Sweep?

In general, a frequency sweep is what you do when checking your favorite radio stations. Set the station for one frequency, and check the output. If you don't like the song that's playing, change the frequency and check the output again.

Activity #1: Testing the Frequency Sweep

The Boe-Bot can be programmed to send different IR frequencies, and to check for object detection at each frequency. By keeping track of the frequencies for which the IR detector reported an object, its distance can be determined. The left axis of the graph in Figure 6.1 shows how the sensitivity of the IR detector's electronic filter decreases as it receives frequencies greater than 38.5 kHz. The filter essentially causes the IR detector to become less able to detect IR at these frequencies. Another way to think about it is that you have to move an object closer if you want it to be detected at a less sensitive frequency. Since the detector is less sensitive, it will take brighter IR (or a closer object) to make the detector see the signal.

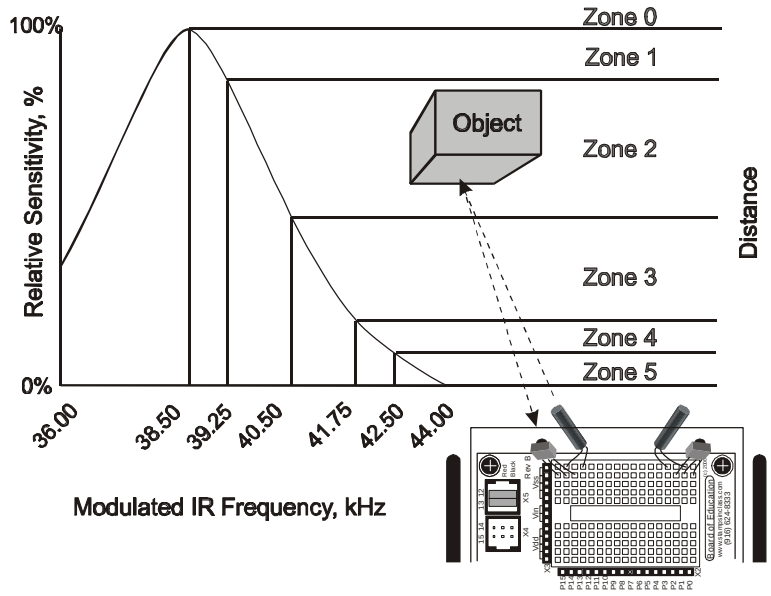


Figure 6.1: The left axis of the graph compares IR frequency to the relative sensitivity of the IR detector. The right side of the graph shows how the relative sensitivity of the IR detector relates to distance detection. As detector sensitivity decreases with the increase in frequency, the object must be closer for the IR signal to be detected. Why closer? When the detectors are made less sensitive by sending higher frequencies, it's like giving them darker and darker lenses to look through. Just as a flashlight beam appears brighter when reflected off an object that's closer to you, IR reflected off a closer object appears brighter to the IR detectors.

The right axis of Figure 6.1 shows how different frequencies can be used to indicate in which zone a detected object is located. By starting with a frequency of 38.5 kHz, whether or not an object is in Zone 1-5 can be determined. If an object is not yet detected, it must be beyond the detector limit (Zone 0). If an object is detected, by testing again at 39.25 kHz, the first datum about distance is collected. If 38.5 kHz is detected the object but 39.25 kHz did not, the object must be in Zone 1. If the object was detected at both frequencies, but not at 40.5 kHz, we know it's in Zone 2. If all three frequencies detected the object, but it was not detected at 41.75 kHz, we know it is in Zone 3. If all four frequencies detected the object, but not 42.5 kHz, we know it's in Zone 4. If all the frequencies detected the object, we know it's in Zone 5.

FYI

The frequency sweep technique used in this chapter works fairly well for the Boe-Bot, and the components are only a fraction of the cost of common IR distance sensors. The trade off is that the accuracy of this method is also only a fraction of the accuracy of common IR distance sensors. For basic Boe-Bot tasks that require some distance perception, such as following another Boe-Bot, this interesting technique does the trick. Along with adding low-resolution distance perception to the Boe-Bot's senses, it also provides an introduction to the concepts of filters and frequency response.

Build It!

- ❑ Use the same IR detection circuit from Chapter 5, shown in Figure 5.4, for this activity.

Programming the IR Distance Gage

Programming the BASIC Stamp to send different frequencies involves a **`for...next`** loop. The **`counter`** variable can be used to give the **`freqout`** command different frequencies to check. This program introduces the use of arrays. Arrays are used in Program Listing 6.1 to store the IR detector outputs at the different frequencies. For the **`l_values`** variable, the Zone 0 output is stored in bit-0 of **`l_values`**. The Zone 1 output is stored in bit-1 **`l_values.bit1`**, and so on, all the way through Zone 5, which is stored in bit-5 of **`l_values`**. The same measurements are taken for **`r_values`**.

- ❑ Enter and run Program Listing 6.1.

- ❑ This program makes use of the Debug Terminal, so leave the serial cable connected to the BOE while Program Listing 6.1 is running.

```
' Robotics! v1.5, Program Listing 6.1: IR Distance Gage.
' {$Stamp bs2}                                ' Stamp Directive.

'----- Declarations --------------

  counter           var          nib          ' Multipurpose counting variable.
  l_values          var          byte         ' Two vars for storing left & right
```

```
  r_values          var          byte          ' freq sweep IR detector outputs.
  IR_freq           var          word          ' Stores frequency arg for freqout.

'----- Initialization ------------

  output 7                                     ' Set all I/O lines sending freqout
  output 1                                     ' signals to function as outputs.

'----- Main Routine --------------

  main:

    l_values = 0                               ' Reset l_values and r_values to 0.
    r_values = 0

    ' Load sensor outputs into l_values and r_values using a for...next loop,
    ' and a lookup table, and bit addressing.

    for counter = 0 to 4

      lookup counter,[37500,38250,39500,40500,41500], IR_freq

      freqout 7,1, IR_freq
      l_values.lowbit(counter) = ~in8

      freqout 1,1, IR_freq
      r_values.lowbit(counter) = ~in0

    next

    ' Display l_values and r_values in binary and ncd format.

    debug home, cr, cr, "Left readings     Right Readings", cr
    debug "  ",bin8 l_values, "          ", bin8 r_values, cr
    debug "  ",dec5 ncd(l_values), "             ", dec5 ncd(r_values), cr, cr

  goto main
```

When the Boe-Bot is placed facing a nearby wall (3 to 5 cm.), the Debug Terminal should display something similar to Figure 6.2. As the Boe-Bot is moved closer to and further from the wall, the numbers displayed by the Debug Terminal should change increase and decrease.

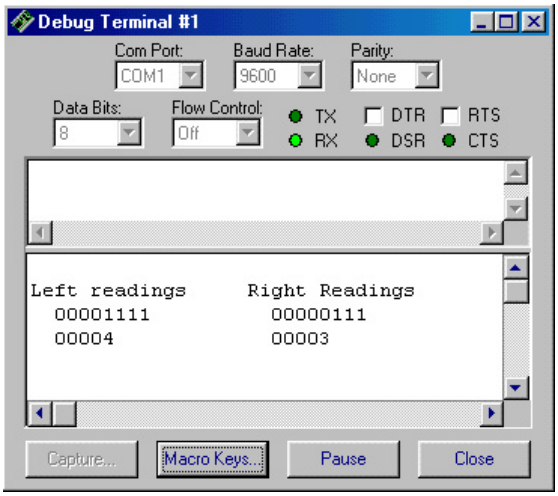


Figure 6.2: Frequency sweep data in binary and ncd format.

- Place the Boe-Bot so that it faces the wall with its IR LEDs about 1 cm. away from the wall. The left and right readings should both be at "4" or "5." If not, make sure each IR detector is facing in the same direction as its IR LED. Also check to make sure you are using 220 Ω resistors.

- Gradually back the Boe-Bot away from the wall. As the Boe-Bot is backed away from the wall, the left and right readings should gradually decrease to "0."

- If either or both sides stay at all zeros or all ones, it indicates a possible mistake in either your wiring or in the program. If this is the case, unplug your battery pack from the BOE. Then, check your wiring and PBASIC code for errors.

FYI	The maximum detection distance is 20 to 30 cm., depending on the reflectivity of the wall. Some tinkering with how far left/right each IR pair is pointing may be required to get the numbers to be the same at a given distance. A high level of precision IS NOT necessary for these activities. Appendix H: Fine Tuning IR Distance Detection presents a method for calibrating the Boe-Bot's distance detectors. This method can be time consuming, and it's not required for Activity #2 or #3.

✓ **TIP**	Use a wire stripper to unsheathe about 1 cm. of insulation from a jumper wire. Slide the insulation up one of the IR LED leads. This will protect the leads from touching each other during adjustment.

How the Distance Gage Program Works

- Look up the **lookup** command in Appendix C: PBASIC Quick Reference or in the BASIC Stamp Manual before continuing.

Counter is a nibble variable that is used to index a **for...next** loop. The **for...next** loop is used for checking the IR detectors at various frequencies. The **l_values** and **r_values** variables store the outputs for the left and right IR detectors at the various frequencies used. Each variable stores five binary measurements. Since the IR detector outputs are tested at a variety of frequencies, **IR_freq** is a variable that can store the value of the frequency that gets sent each time through the frequency testing loop.


```
declarations:

  counter         var         nib
  l_values        var         byte
  r_values        var         byte
  IR_freq         var         word
```

The main routine contains two routines, one for frequency sweep and another for displaying the data collected. The first step in the frequency sweep is setting **l_values** and **r_values** to zero. This is important since individual bits in each variable are modified. Clearing **l_values** and **r_values** starts each variable with a clean slate. Then individual bits can be set to "1" or "0," depending on what the IR detectors report.

```
main:

  l_values = 0
  r_values = 0
```

The **for…next** loop is where the frequency sweep occurs. The **lookup** command checks the **counter** value to determine which frequency to copy to the **IR_freq** variable. When **counter** is "0," 37500 gets copied to **IR_freq**. When **counter** is "1," 38250 is copied to **IR_freq**. As the value of **counter** is incremented from "0" to "4" by the **for...next** loop, each successive value in the **lookup** table is copied to **IR_freq**.

```
  for counter = 0 to 4

     lookup counter,[37500,38250,39500,40500,41500],IR_freq
```

Note that the lookup table begins the frequency sweep at 37500 (most sensitive) and ends at 41500 (least sensitive). You might be wondering why the numbers in the lookup table don't match the frequency values from Figure 6.1. It's true that if the BASIC Stamp could transmit a 50% duty cycle pulse train (pulses with the same high time and low time) at these frequencies, they would have to match the frequencies specified for the IR detector's filter. However, the **freqout** command introduces other factors that affect the amplitude of the harmonics transmitted by the IR LEDs. The math involved in predicting the optimum ***frequency*** arguments to use is very advanced and is well outside the scope of this text. Even so, the best frequencies for a given distance can be determined experimentally. You can consult Appendix H: Tuning IR Distance Detection for details, but try the remaining activities in this book first because the list of values we are using are known to be reliable.

The left sensor is checked by using **freqout** to send the current value of **IR_freq**. Next, the **.lowbit()** argument is used to address each successive bit in **l_values**. When **counter** is "0," the **.lowbit(counter)** argument addresses bit-0 of **l_values**. When **counter** is "1," the **.lowbit(counter)** argument addresses bit-1 of **l_values**, and so on. Before writing the value of **in8** to **l_values.lowbit(counter)**, the NOT operator (~) is used to invert the bit's value before it is stored to its bit array location in **l_values**. The same process is then repeated for **r_values**. After the fifth time through the **for...next** loop, the IR data bits have all been loaded into **l_values** and **r_values**.

```
            freqout 7,1,IR_freq
            l_values.lowbit(counter) = ~in8

            freqout 1,1,IR_freq
            r_values.lowbit(counter) = ~in0

        next
```

The **display** subroutine uses a variety of formatters and text strings to display the **l_values** and **r_values** variables. The first row of the display is the text heading indicating which readings correspond the right IR detector and which readings correspond to the left IR detector. Remember that left and right are treated as though you are sitting in the Boe-Bot driver's seat.

```
display:
  debug home, cr, cr, "Left readings     Right Readings", cr
```

The second row displays **l_values** and **r_values** in binary format. This allows for observation of how the bit values in **l_values** and **r_values** change as the apparent distance of an object changes.

```
  debug "  ",bin8 l_values, "          ", bin8 r_values, cr
```

The third row displays the **ncd** value of each variable. The **NCD** operator returns a value that corresponds to the location of the most significant bit in a variable. If the variable is all zeros, **ncd** returns a zero. If the least significant bit contains a "1," and all the rest of the digits are "0," **NCD** returns a "1." If bit-1 contains a "1," but all the numbers to the left of bit-1 are zeros, **ncd** returns a "2," and so on. The **NCD** operator is a handy way of indicating how many ones have been loaded into the lower bits of **l_values** and **r_values**. What's really handy is that **ncd** directly tells you in which zone the object has been detected.

```
debug "  ",dec5 ncd(l_values), "           ", dec5 ncd(r_values), cr, cr
```

When the display routine is finished sending data to the Debug Terminal, program control is returned to the **main** label.


```
goto main
```

Your Turn

- With Program Listing 6.1 running, place the Boe-Bot facing the wall so that the IR LEDs are about 1.5 cm. from the wall. For best results, tape a white sheet of paper to the wall.
- Make a note of the left and right readings.
- Start pulling the Boe-Bot away from the wall.
- Each time the value of one or the other sensors decreases, make a note of the distance. In this way you can determine the zones for each of your Boe-Bot's IR pairs.
- If the readings on one side are consistently larger than the other, you can point the IR LED on the side reporting the larger readings outward a little further. For example, if the left IR pair continually reports higher readings than the right IR pair, try pointing the left IR LED and detector a little further to the left.

Activity #2: The Drop-off Detector

One application for distance detection is checking for a drop-off. For example, if the Boe-Bot is navigating on a table, it can change direction if it sees the edge of the table. All you have to do is point the IR pairs downward so that they are both pointing at the table right in front of the Boe-Bot. A distance detection program can then be used to detect that the table is close-up. When the Boe-Bot nears the edge of a table, one or both of the distance detectors will start reporting that they no longer see something close-up. That means it's time to turn away from the abyss.

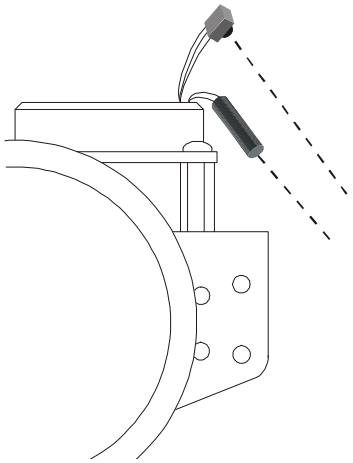


Figure 6.3: IR LED adjustment for edge detection.

- ❑ Point your IR pairs at the surface directly in front of the Boe-Bot as shown in Figure 6.3. The IR pairs should be pointed downward 45° from horizontal and outward 45° from the Boe-Bot's center line.
- ❑ Perform the tests below using Program Listing 6.1 before trying program Listing 6.2.

> ✓ **TIP** Go through the Chapter #6 activities first using the numbers given in the program listings. If you used Appendix H to calibrate your Boe-Bot's distance detectors, try the values you determined only after first trying the values given in the example programs.

- ❑ Record the IR pair outputs when the Boe-Bot is looking straight at the table. If the values of the IR pairs when they are looking at your tabletop are "3" or more, it indicates your detectors are seeing what they are supposed to see.
- ❑ Record the IR pair outputs when the Boe-Bot is looking off the edge of the table. If these values remain less than "3," the Boe-Bot is ready to try Program Listing 6.2.
- ❑ If the Boe-Bot does not give you steady and consistent readings of "3" or more when the Boe-Bot is looking at the table, try first adjusting the direction the IR pairs are pointing. Also, if the Boe-Bot does not consistently register less than "3" when it's looking off the edge of the table, some additional adjustment of the IR pairs also is in order.

- If the sensors report "3" or more while looking at the table and "2" or less when looking off the edge, the Boe-Bot is ready for Program Listing 6.2.

- If no physical adjusting works, try the instructions in Appendix H, then repeat the tests above.

Make sure to be the spotter for your Boe-Bot when running Program Listing 6.2. Always be ready to pick your Boe-Bot up 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!

When spotting your Boe-Bot while it's avoiding drop-offs, be ready to pick it up from above. Otherwise, the Boe-Bot will see your hands instead of the drop-off and not perform as expected..

Programming for Drop-Off Detection

Program Listing 6.2 uses modified versions of the forward, right_turn, left_turn and backward routines that have been used and reused in every chapter since Chapter #2. The number of pulses in each routine have been adjusted for better performance along a table edge. The **check_sensors** subroutine takes distance measurements by recycling code from Program Listing 6.1: IR Distance Gage.

- Run and test Program Listing 6.2. Remember, always be ready to pick your Boe-Bot up if it tries to run off the table.

```
' Robotics! v1.5, Program Listing 6.2: Drop-off Detection
' {$Stamp bs2}                          ' Stamp Directive.

'----- Declarations --------------

  counter     var    nib                 ' For...next loop index variable.
  l_values    var    word                ' Store R sensor vals for processing.
  r_values    var    word                ' Store L sensor vals for processing.
  l_IR_freq   var    word                ' Stores L IR freqs from lookup table.
  r_IR_freq   var    word                ' Stores R IR freqs from lookup table.

'----- Initialization ------------

   low 13                                ' Initialize servo line startup values.
   low 12
   output 2                              ' Declare freqout lines to be outputs.
   output 7
```

```
   output 1
   freqout 2,500,3000                  ' Signal program is starting/restarting.

'----- Main Routine --------------

main:                                  ' Main routine

  ' The command “gosub check_sensors” sends the program to a subroutine that
  ' loads distance values into l_values and r_values.  So, when the program returns
  ' from the check_sensors subroutine, the values are updated and ready for
  ' distance based decisions.

  gosub check_sensors

  ' The distances are checked for four different inequalities.  Depending on the
  ' inequality that turns out to be true, the program either branches to the
  ' forward, left_turn, right_turn or backward navigation routine.

  if l_values >= 3 and r_values >= 3 then forward
  if l_values >= 3 and r_values < 3 then left_turn
  if l_values < 3 and r_values >= 3 then right_turn
  if l_values < 3 and r_values < 3 then backward

goto main                              ' Repeat the process.

'----- Navigation Routines -------

   forward:                            ' Deliver a single forward pulse, then
      pulsout 13,1000
      pulsout 12,500
      pause 10
   goto main                           ' go back to the main: label.

   left_turn:                          ' Deliver eight left pulses, then
      for counter = 0 to 8
         pulsout 13,500
         pulsout 12,500
         pause 20
      next
   goto main                           ' go back to the main: label.

   right_turn:                         ' Deliver eight right pulses, then
      for counter = 0 to 8
         pulsout 13,1000
         pulsout 12,1000
         pause 20
      next
```

```
   goto main                            ' go back to the main: label.

   backward:                            ' Deliver eight backward pulses, then
      for counter = 0 to 8
         pulsout 13,500
         pulsout 12,1000
         pause 20
      next
   goto main                            ' go back to the main: label.

'----- Subroutines ---------------

  ' The check sensors subroutine is a modified version of Program Listing 6.1
  ' without the Debug Terminal display.  Instead of displaying l_values and
  ' r_values, the main routine uses these values to decide which way to go.

  check_sensors:

    l_values = 0                                ' Reset l_values and r_values to 0.
    r_values = 0

    ' Load sensor outputs into l_values and r_values using a for...next loop,
    ' a lookup table, and bit addressing.

      for counter = 0 to 4

         check_left_sensors:
            lookup counter,[37500,38250,39500,40500,41500],l_IR_freq
            freqout 7, 1, l_IR_freq
            l_values.lowbit(counter) = ~ in8

         check_right_sensors:
            lookup counter,[37500,38250,39500,40500,41500],r_IR_freq
            freqout 1, 1, r_IR_freq
            r_values.lowbit(counter) = ~ in0

      next

    ' Convert l_values and r_values from binary to ncd format.

      l_values = ncd l_values
      r_values = ncd r_values
```

```
    ' Now l_values and r_values each store a number between 0 and 5 corresponding
    ' to the zone the object is detected in.  The program can now return to the
    ' part of the main routine that makes decisions based on these distance
    ' measurements.

   return
```

How the Drop-off Avoidance Program Works

The first thing the **main** routine does is call the **check_sensors** subroutine. Note that **check_sensors** is simply Program Listing 6.1 with no Debug Terminal display placed in a subroutine. Instead of debugging the **NCD** values of **l_detect** and **r_detect**, the values of these two variables are simply converted to **ncd** values using the statements:

```
l_values = ncd l_values
```

and

```
r_values = ncd r_values
```

After calling the **check_sensors** subroutine, **l_values** and **r_values** are numbers between "0" and "5." These values are used instead of the "1" and "0" values used in the whiskers program. After the program returns from the **check_sensors** subroutine, **l_values** and **r_values** are checked against the benchmarks distance indicating the edge of the table has been detected.

```
if l_values >= 3 and r_values >= 3 then forward
if l_values >= 3 and r_values < 3 then left_turn
if l_values < 3 and r_values >= 3 then right_turn
if l_values < 3 and r_values < 3 then backward
```

The ***start*** and ***end*** arguments in the **for...next** loops that deliver the pulses in the four navigation routines have been modified slightly. They have also been relocated to a **Navigation Routines** section. This makes adjustment of the routines' behavior easier because they are all in one place.

Your Turn

- By further adjusting the ***start*** and ***end*** arguments of the **for...next** loops in the navigation routines, you can program a variety of drop-off avoidance behaviors.

Activity #3: 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 the lead vehicle is ahead. 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.

Proportional control can be used to accomplish this by taking the difference between the desired distance and the measured distance and then adjusting the pulse widths sent to the servos accordingly. Figure 6.4 shows how proportional control works. The measured distance is subtracted from the desired distance, which is called the set point. By subtracting the measured distance from the set point distance, you get the distance error. Adjusting the distance to make the measured distance equal to the set point distance involves multiplying the error by a number, called the proportionality constant, K_p.

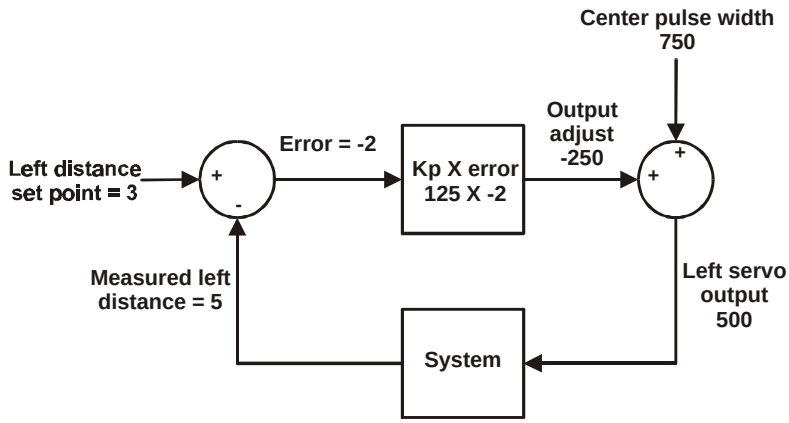


Figure 6.4: Boe-Bot proportional control block diagram.

Figure 6.4 also illustrates a specific example for the left servo and IR pair. The set point is "3;" the distance is "5," and the proportionality constant is $K_p = 125$. Starting with the circle at the left of the block diagram, which is called a summing junction, the measured distance is subtracted from the set point distance. Since the set point distance is "3" and the measured distance is "5," the error value is –2. An error signal of –2 means the Boe-Bot shadow vehicle has to back away from the lead vehicle because the lead vehicle is just too darn close! The block the error value feeds into multiplies the error by $K_p = 125$. So the block performs the following operation: $125 \times (-2) = -250$. The summing junction on the right adds –250 to the left servo's center pulse width. The result is that 100 is subtracted from the left servo's center value. The net effect is a pulse width of 500 to the left servo, which is full-speed backward. The math for this proportional control operation is fairly straightforward.

$$\begin{aligned} \text{left_width} &= \text{center} + (K_p \times \text{error}) \\ &= 750 + (125 \times (-2)) \\ &= 500 \end{aligned}$$

The right servo and IR pair have a similar algorithm, except that the value of the output adjust is subtracted instead of added to the center pulse width ***period*** of 750 (1.5 ms). Assuming the same measured value at the right IR pair, the output adjust results in a pulse width of 1000. The end result is that the Boe-Bot applies a full speed reverse pulse. This reverse pulse is part of a complex system that depends on unknown factors relating to the lead vehicle's motion. The idea of feedback is that the system's output is resampled, by the shadow Boe-Bot taking another distance measurement. Then the control loop repeats itself again and again and again…

Programming the Boe-Bot Shadow Vehicle

Program Listing 6.3 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 K_p, and the resulting value is added/subtracted to/from the pulse widths to the left/right servos.

- ❑ Run Program Listing 6.3.
- ❑ 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.

```
' Robotics! v1.5, Program Listing 6.3: Shadow Vehicle
' {$Stamp bs2}                    ' Stamp Directive.

'----- Declarations --------------
                                  ' Constants
  Kp_r        con          125    '     Right servo proportional constant.
  Kp_l        con          125    '     Left servo proportional constant.
  set_point   con          3      '     Set distance to a value between 0 & 5.
                                  ' Variables
  counter     var    nib           '     For...next loop index variable.
  l_values    var    word          '     Store L sensor vals for processing.
  r_values    var    word          '     Store R sensor vals for processing.
  l_IR_freq   var    word          '     Stores L IR freqs from lookup table.
  r_IR_freq   var    word          '     Stores R IR freqs from lookup table.
```

```
'----- Initialization ------------

   output 13                      ' Declare outputs.
   output 12
   output 2
   output 7
   output 1

   freqout 2,500,3000             ' Beep at startup.

'----- Main Routine --------------

main:                             ' Main routine

  gosub check_sensors             ' Get distance values for each sensor

  l_values = kp_l * (set_point - l_values)     ' Left proportional control.
  r_values = kp_r * (set_point - r_values)     ' Right proportional control.

  pulsout 13,750 + l_values       ' Pulse servos
  pulsout 12,750 - r_values

goto main                         ' Infinite loop.

'----- Subroutine(s) -------------

check_sensors:

  l_values = 0                    ' Set distances to 0.
  r_values = 0
  ' Take 5 measurements for distance at each IR pair.  If you fine tuned your
  ' frequencies in Activity #2, insert them in the lookup tables.

  for counter = 0 to 4
    check_left_sensors:
      lookup counter,[37500,38250,39500,40500,41000],l_IR_freq
      freqout 7,1,l_IR_freq
      l_values.lowbit(counter) = ~in8

    check_right_sensors:
      lookup counter,[37500,38250,39500,40500,41000],r_IR_freq
      freqout 1,1,r_IR_freq
      r_values.lowbit(counter) = ~in0

  next
```

```
  l_values = ncd l_values        ' Value (0 to 5) for distance depending on MSB.
  r_values = ncd r_values

return
```

How the Shadow Vehicle Program Works

Program Listing 6.3 declares three constant aliases, **Kp_r**, **Kp_l**, and **set_point** using the **con** directive. Everywhere you see **set_point**, it's actually the number "3" (a constant). Likewise, everywhere you see either **Kp_r** or **Kp_l** in the program, the value is actually the number 125. The convenient thing about declaring aliases for these constants is that when the alias declaration is changed, all instances of the alias are changed throughout the entire program. For example, by changing the **Kp_l con** directive from 125 to 100, every instance of **Kp_l** in the entire program changes from 125 to 100. This is exceedingly useful for experimenting with and tuning the right and left proportional control loops.

```
declarations:
  Kp_r            con         125
  Kp_l            con         125
  set_point       con         3
```

The first thing the **main** routine does is call the **check_sensors** subroutine. After the **check_sensors** subroutine is finished, **l_values** and **r_values** each contain a number corresponding to the zone in which an object was detected for both the left and right IR pairs.

```
main:

  gosub check_sensors
```

The next two lines of code implement both error and proportional calculations for each servo. The error calculation for each side is the subtraction within the parenthesis and the proportional is the multiplication of **Kp_l** and **Kp_r** by the terms in parenthesis.

```
  l_values = kp_l * (set_point - l_values)
  r_values = kp_r * (set_point – r_values)
```

The output adjust or drive calculations are nested in the **pulsout *period*** arguments. No **pause *period*** was included because it takes 10 ms (plus some processing time) just to run the 10 frequency sweep values.

```
  pulsout 13,750 + l_values
  pulsout 12,750 - r_values
```

Then program control is returned to the **`main:`** label, and the proportional feedback loop repeats itself.

```
goto main
```

Your Turn

Figure 6.5 shows a lead Boe-Bot followed by a shadow Boe-Bot. The lead Boe-Bot is running a modified version of Program Listing 5.3: IR Roaming, and the shadow Boe-Bot is running Program Listing 6.3: Shadow Vehicle. Proportional control makes the shadow Boe-Bot a very faithful follower. One lead Boe-Bot can string along a chain of 6 or 7 Boe-Bots. Just add the lead Boe-Bot's side panels and tailgate to the rest of the shadow Boe-Bots in the chain.

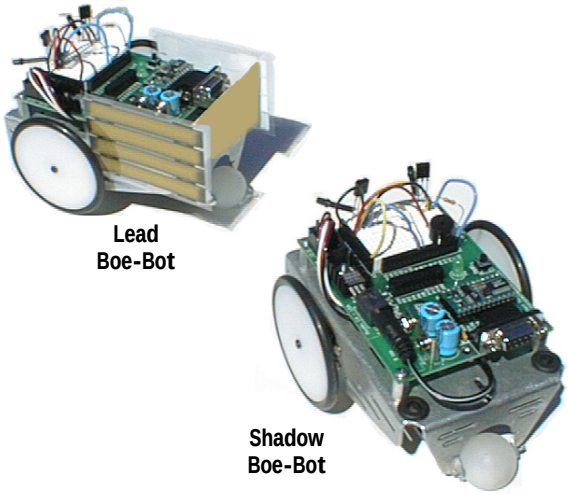


Figure 6.5: Lead and shadow Boe-Bots.

- 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 6.5.
- 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.
- Program the lead Boe-Bot for object avoidance using Program Listing 5.3 with these modifications to the four **`Navigation Routines`**:
  - Increase all pulse width ***`period`*** arguments that are 500 to 650.
  - Reduce all pulse width ***`period`*** arguments that are 1000 to 850.
- The Shadow Boe-Bot should be running Program Listing 6.3 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 follows 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 Program Listing 6.3 using values of **`kp_r`** and **`kp_l`** constants, ranging from 50 to 150. Note the difference in how responsive the Boe-Bot is when following an object.

- Try making adjustments to the value of the **`set_point`** constant. Try values between from 1 to 4. This will have a significant effect on the Boe-Bot's tendency to correct forward or backward. When set_point is 1 or 4, **`kp_r`** and **`kp_l`** should be about 60. When **`set_point`** is 2 or 3, **`set_point`** can be 125.

Summary and Applications

A technique frequently used in electronics called frequency sweep was introduced. The Boe-Bot was programmed to run a frequency sweep on the IR detectors, which have built-in bandpass filters. Each bandpass filter has a center frequency of 38.5 kHz. The frequency response of each bandpass filter was used to indicate the distance of objects detected by the filter.

Closed-loop feedback and proportional control also were introduced. 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 perform this operation in discrete time by taking repeated samples and recalculating the output adjust. Every 20 ms, the Boe-Bot re-sampled the distance and calculated a servo output proportional to the error signal, which was the desired distance subtracted from the measured distance.

Real World Examples

Filters are used in a variety of applications from radio, television, cell phone communication, and high-fi audio to, of all things, IR communication. A wide variety of filters are available with many different frequency response characteristics. Mechatronics, analog and digital electronics, and even mechanics feature many electronic filter applications and design techniques. One mechatronics example of where a filter would come in handy is a vibration detector. Assuming certain frequencies of vibration could harm the moving parts of a machine, a sensor feeding electronic sensor output to a filter could be used to always look for vibration at this frequency. When the vibration is detected, the filter passes the signal through to a subsystem that applies automatic correction.

Speaking of automatic correction, proportional control is the beginning of an entire class of closed-loop feedback control applied to a variety of industrial, electronic, and mechatronic systems. On-off control is the first line of defense in keeping a system such as a fluid tank, a pressure system or a temperature system at a preset level. Proportional control is the second line of defense. Then comes various mixtures of proportional, integral and derivative control introduced in the Stamps in Class Industrial Control text, available from www.stampsinclass.com.

Boe-Bot Application

IR object detection can now be used to determine distance as well as presence of an object. Add this feature to detection of light intensity, and tactile detection of objects, and the Boe-Bot's ability to perceive its world is significantly increased. This text also introduced a variety of navigation techniques for following various things such as lines, lights, and other Boe-Bots. Navigation techniques for making the Boe-Bot flee from

other things, such as obstacles and drop-offs, also were introduced. Most importantly, this text demonstrated ways of reusing navigation and sensory programming techniques introduced. The way these techniques were used and incorporated into PBASIC programs depended on the situation at hand. The important feature is that you are now ready to try mixing and matching sensor and navigation techniques in a Boe-Bot competition. See Appendix I: Boe-Bot Competition Maze Rules.

A variety of other sensor, electronic and control system techniques are introduced in other Stamps in Class texts. If you're interested in adding to your bag of Boe-Bot tricks or have plans to compete in the Micromouse, Sumo Robot, or Firefighting Robot competitions, you'll be pleased with the additional techniques and skills introduced in these texts, all available for free download from www.stampsinclass.com:

- What's a Microcontroller: The basics of microcontrolled electronic projects with the BASIC Stamp. PBASIC programming is used to interface the BASIC Stamp with elementary sensors and peripheral devices.

- Basic Analog and Digital: The BASIC Stamp is becoming increasingly popular as the brain of many intelligent electronic designs. This text introduces a variety of analog-to-digital and digital-to-analog conversion techniques that lend themselves to use with the BASIC Stamp. Also introduced are some of the fundamental communication protocols for use with peripheral converters.

- Earth Measurements: The BASIC Stamp can also be used at the heart of many environmental and data logging devices. This text delves further into the world of intelligent electronic sensors and the physics of accurate environmental measurements. Along the way, a variety of useful and clever BASIC Stamp sensor interfaces are introduced. These concepts then are applied to microcontrolled environments. The examples included lend themselves to further exploration into weather, hydroponics, and a variety of other scientific endeavors.

- Industrial Control: This text uses a variety of BASIC Stamp projects as a launching pad into fundamental principles and concepts in industrial process control. Free downloadable StampPlot Lite software adds a unique oscilloscope type of data collection to the projects, and is particularly useful for illustrating system response to BASIC Stamp control using the following techniques: open loop, differential gap, proportional, integral and derivative.

Questions and Projects

Questions

1. What effect does the IR detector's electronic filter have on its ability to detect IR reflected off objects?

2. How does the IR detector's electronic filter make it possible to detect an object's distance?

3. What kind of data is necessary to fine tune the frequency sweep? What is the input variable that's varied? What is the output data that's examined? What decisions are made based on the output data?

4. What variable declaration and looping considerations would be involved in setting up a 16-value frequency sweep for each IR pair?

5. How can distance detection be used to detect the edge of a table?

6. How can the Boe-Bot use distance detection to maintain a constant distance from a stationary object? Is any different programming necessary for the Boe-Bot to maintain a constant distance from a moving object?

7. Describe the proportional control feedback loop. How does this loop apply to the Boe-Bot for the shadow vehicle exercise?

Exercises

1. If seven zones are needed, determine how many test frequencies will be required.

2. How many test frequencies would be required for 16 zones?

3. Modify Program Listing 6.3 so that the Boe-Bot can test for 16 zones per IR pair. Adjust this code so that it maintains a 20 ms pause between servo pulses.

4. Write a segment of code that will allow the Boe-Bot to lock onto and follow a table edge. Hint: proportional control will help in this exercise.

Projects

1. Program the Boe-Bot to follow a black stripe on a white background. Hint: A black stripe looks the same to the IR sensors as a far away object does. This will involve modifying Program Listing 6.2 so that it's attracted to far-away objects.

2. Add ramping algorithms to the Program Listings 6.2 and 6.3, and test them.

3. Program the Boe-Bot to follow the edge of a black stripe. This is different from following a black stripe with both detectors. The Boe-Bot should be looking for a transition from black to white. When you're done, the Boe-Bot should be able to lock onto and follow a black stripe on either its left or right side.

4. If you succeeded in Project #3, try making a maze that's delimited by black stripes. Hint: EEPROM navigation will help in certain maneuvers like navigating corners. The cheese in the maze is a flashlight pointed at the white surface. When the Boe-Bot detects the cheese, it should stop and flash an LED on and off 10 times. A good cheese detector would be a single photoresistor circuit.

5. Challenge: Whiskers can be mounted on the Boe-Bot, and you can train it to better navigate the black and white delimited maze in this chapter's Project #4. The foundation for this training is introduced in Chapter #3, Project #3.

Appendix A: Boe-Bot Parts Lists and Sources

Robotics! - Full Kit

Aside from a PC and the simple tools listed on Page 7, the Robotics! Full Kit contains all the parts and documentation you'll need to complete the experiments in this text.

Robotics! – Full Kit (#28132)

Parallax Code#	Description	Quantity
BS2-IC	BASIC Stamp II module	1
27919	BASIC Stamp Manual Version 1.9	1
28124	Robotics! Parts Kit	1
28125	Robotics! v1.5 text	1
550-00010	Board of Education Rev B	1
800-00003	Serial cable	1

Robotics! - Parts Kit

If you already have a BOE and BASIC Stamp, the Robotics! Parts Kit can also be purchased separately. As with all Stamps in Class curriculum, the Robotics! experiments are designed for use with the Board of Education with BASIC Stamp and the parts kit with the same name as the text. The contents of the Robotics! Parts Kit is listed below. Replacement parts in the kit may also be ordered from http://www.stampsinclass.com.

Robotics! Parts Kit (#28124)

Parallax Code#	Description	Quantity
150-01030	10K Ω resistors	2
150-02210	220 Ω resistors	6
150-04710	470 Ω resistors	4
200-01031	0.01 μF capacitors	2
200-01040	0.1 μF capacitors	2
201-03080	3300μF capacitor	1
350-00006	red LEDs	2
350-00009	Photoresistors (EG&G Vactec VT935G group B)	2
350-00014	infrared receiver (Panasonic PNA4602M or eq.)	2
350-00017	infrared LEDs covered with heat shrink tubing (QT QEC113)	2

Robotics! Parts Kit (#28124 continued)

Parallax Code#	Description	Quantity
451-00303	3-Pin Header	4
700-00010	Whisker	2
700-00015	#4 screw-size nylon washer	2
700-00049	3/8" Male-female standoff	2
800-00016	jumper wires (bag of 10)	2
900-00001	Piezospeaker	1
28133	**Boe-Bot Hardware Pack**	
700-00002	4-40 x 3/8"machine screws	8
700-00009	1" polyethylene ball, pre-drilled	1
700-00011	o-ring tires	2
700-00013	Boe-Bot plastic machined wheels	2
700-00016	4-40 x 3/8" flathead machine screws	2
700-00022	Boe-Bot aluminum chassis	1
700-00023	1/16" x 1.5" long cotter pin	1
700-00024	4-40 locknuts	10
700-00025	13/32" rubber grommet (fits ½" hole)	1
700-00026	9/32" rubber grommet (fits 3/8" hole)	2
700-00027	1/2" double-female standoffs	4
700-00028	4-40 x 1/ 4" machine screw	8
700-00038	Battery holder with cable and barrel plug	1
900-00008	Parallax pre-modified servos	2

Board of Education Kits

The Stamps in Class curricula features different modules that depend on the BASIC Stamp and Board of Education as a core. This core can be purchased separately.

Board of Education – Full Kit (#28102)

Parallax Code#	Description	Quantity
550-00008	Board of Education Rev A	1
800-00016	Pluggable wires	10
BS2-IC	BASIC Stamp II module	1
750-00008	300 mA 9 VDC power supply	1
800-00003	Serial cable	1

Board of Education Kit (#28150)

Parallax Code#	Description	Quantity
550-00008	Board of Education Rev A	1
800-00016	Pluggable wires	6

Robotics! and BASIC Stamp Documentation

The documentation included with the Robotics! Full Kit is also available separately.

Robotics! Documentation

Parallax Code#	Description	Internet Availability?
27919	BASIC Stamp Manual Version 1.9	http://www.stampsinclass.com
28125	Robotics! v1.5	http://www.stampsinclass.com

Parallax Distributors

The Parallax distributor network serves approximately 40 countries worldwide. A portion of these distributors are also Parallax-authorized "Stamps in Class" distributors – qualified educational suppliers. Stamps in Class distributors normally stock the BASIC Stamp and Board of Education (#28102 and #28150). Several electronic component companies are also listed for customers who wish to assemble their own Robotics Parts Kit.

Country	Company	Notes
United States	**Parallax, Inc.** 3805 Atherton Road, Suite 102 Rocklin, CA 95765 USA (916) 624-8333, fax (916) 624-8003 http://www.stampsinclass.com http://www.parallaxinc.com	Parallax and Stamps in Class source. Manufacturer of the BASIC Stamp.
United States	**Digi-Key Corporation** 701 Brooks Avenue South Thief River Falls, MN 66701 (800) 344-4539, fax (218) 681-3380 http://www.digi-key.com	Source for electronic components. Parallax distributor. May stock Board of Education. Excellent source for components.
Australia	**Microzed Computers** PO Box 634 Armidale 2350 Australia Phone +612-67-722-777, fax +61-67-728-987 http://www.microzed.com.au	Parallax distributor. Stamps in Class distributor. Excellent technical support.

Appendix A: Boe-Bot Parts Lists and Sources

Australia	**RTN** 35 Woolart Street Strathmore 3041 Australia Phone / fax +613 9338-3306 http://people.enternet.com.au/~nollet	Parallax and Stamps in Class distributor.
Canada	**HVW Technologies** 300-8120 Beddington Blvd NW, #473 Calgary, AB T3K 2A8 Canada (403) 730-8603, fax (403) 730-8903 http://www.hvwtech.com	Parallax distributor and Stamps in Class distributor.
Germany	**Elektronikladen** W. Mellies Str. 88 32758 Detmold Germany 49-5232-8171, fax 49-5232-86197 http://www.elektronikladen.de	Parallax distributor and Stamps in Class distributor.
New Zealand	**Trade Tech** Auckland Head Office, P.O. Box 31-041 Milford, Auckland 9 New Zealand +64-9-4782323, fax 64-9-4784811 http://www.tradetech.com	Parallax distributor and Stamps in Class distributor.
United Kingdom	**Milford Instruments** Milford House 120 High St., S. Milford Leeds YKS LS25 5AQ United Kingdom +44-1-977-683-665 Fax +44-1-977-681-465 http://www.milinst.demon.co.uk	Parallax distributor and Stamps in Class distributor.

Appendix B: PC to Stamp Communication Trouble-shooting

When Identify is selected from the Stamp Editor's Run menu, the expected response is:

"Information: Found BS2-IC (firmware v1.0.)."

When you get an error message instead, it can sometimes be mystifying. Here are descriptions of the most common error messages along with some suggestions for remedying the problems associated with them. If none of these remedies work, go back and make sure all the other instructions in Chapter #1 were followed correctly. Still no luck? Parallax Technical Support contact information is printed on the BOE.

"Error: Basic Stamp II detected but not responding...Check power supply." The software thinks it found a BASIC Stamp on one of the com. ports, but when it tries to communicate with the BASIC Stamp, it gets no response. "Check power supply" means check your battery pack and batteries.

- For troubleshooting, make sure you are using new AA Alkaline 1.5 V batteries and that they are loaded properly into the battery pack.
- Also, make sure the barrel plug is seated into the barrel jack on the BOE.

Is the BASIC Stamp connected to the com. port the software thinks it sees it on?

- To check and adjust your com. port settings, click Edit, and select Preferences.
- Next, click the Editor Operation tab.
- The Default Com. port can be set to different values depending on which com port you think the BASIC Stamp is connected.

There is also an AUTO setting that causes the software to search for a BASIC Stamp on all known com ports.

- After each adjustment in the Default com port settings, click OK, then try Run | Identify again.

"Error: BASIC Stamp II not responding...Check serial cable connection. Check power supply." This message means that the software can't find the BASIC Stamp. The message might state that the BASIC Stamp was not found on a particular com port, or it might say it can't find the BASIC Stamp on any com port.

- Check to see if your serial cable is properly connected.

- Try adjusting the com port settings as outlined for the previous error message.

"Error: BASIC Stamp IISX not responding…Check serial cable connection. Check power supply." Alternatively, this message might specify the BASIC Stamp II E, or P. This message means that the software is set to look for a BASIC Stamp 2SX (or 2E or 2P) instead of a BASIC Stamp 2. This is easy to fix.

- Click Edit and select Preferences.
- Click the Editor Operation Tab.
- Change the Default Stamp Mode field so that it reads BS2 instead of BS2SX or BS2E or BS2P.

Appendix C: PBASIC Quick Reference

The Parallax BASIC Stamp Manual Version 1.9 consists of approximately 450 pages of PBASIC command descriptions, application notes, and schematics. The entire document is available for download from http://www.parallaxinc.com and http://www.stampsinclass.com in Adobe's PDF format, but also may be purchased by students and educational institutions.

This PBASIC Quick Reference Guide is a reduced version of BASIC Stamp II commands.

BRANCH
BRANCH *offset*, **[***address0, address1,…,address N***]**
Go to the address specified by offset (if in range).

- *Offset* is a variable / constant that specifies which of the listed address to go to (0-N).
- *Addresses* are labels that specify where to go.

BUTTON
BUTTON *pin, downstate, delay, rate, workspace, targetstate, label*
Debounce button, perform auto-repeat, and branch to address if button is in target state.

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *downstate* is a constant, expression or a bit, nibble, byte or word variable in the range 0..1.
- *delay* is a constant, expression or a bit, nibble, byte or word variable in the range 0..255.
- *rate* is a constant, expression or a bit, nibble, byte or word variable in the range 0..255.
- *workspace* is a byte or word variable.
- *targetstate* is a constant, expression or a bit, nibble, byte or word variable in the range 0..1.
- *label* is a valid label to jump to in the event of a button press.

COUNT
COUNT *pin, period, result*
Count cycles on a pin for a given amount of time (0 - 125 kHz, assuming a 50/50 duty cycle).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *period* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535.
- *result* is a bit, nibble, byte or word variable.

DATA
DATA {*pointer***} DATA {@***location***,} {***WORD***} {***data***}{(***size***)} {, {** *WORD***} {***data***}{(***size***)}…}**
Store data in EEPROM before downloading PBASIC program.

- *pointer* is an optional undefined constant name or a bit, nibble, byte or word variable which is assigned the value of the first memory location in which data is written.
- *location* is an optional constant, expression or a bit, nibble, byte or word variable which designates the first memory location in which data is to be written.
- *word* is an optional switch which causes DATA to be stored as two separate bytes in memory.
- *data* is an optional constant or expression to be written to memory.

DEBUG
DEBUG *outputdata***{,***outputdata***...}**
Send variables to PC for viewing.

- *outputdata* is a text string, constant or a bit, nibble, byte or word variable. If no formatters are specified DEBUG defaults to ascii character display without spaces or carriage returns following the value.

DTMFOUT
DTMFOUT *pin***, {***ontime***,** *offtime***,}[***key***{,***key***...}]**
Generate DTMF telephone tones.

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *ontime* and *offtime* are constants, expressions or bit, nibble, byte or word variables in the range 0..65535.
- *key* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

END
END

- Sleep until the power cycles or the PC connects. Power consumption is reduced to approximately 50 µA.

FOR...NEXT
FOR *variable* **=** *start* **to** *end* **{***stepVal***} ...NEXT**
Create a repeating loop that executes the program lines between FOR and NEXT, incrementing or decrementing variable according to stepVal until the value of the variable passes the end value.

- *Variable* is a bit, nib, byte, or word variable used as a counter.
- *Start* is a variable or constant that specifies the initial value of the variable.
- *End* is a variable or constant that specifies the end value of the variable. When the value of the variable passes end, the FOR . . . NEXT loop stops executing and the program goes on to the instruction after NEXT.
- *StepVal* is an optional variable or constant by which the variable increases or decreases with each iteration through the FOR / NEXT loop. If start is larger than end, PBASIC2 understands stepVal to be negative, even though no minus sign is used.

FREQOUT
FREQOUT *pin*, *milliseconds*, *freq1* **{**,*freq2***}**
Generate one or two sine waves of specified frequencies (each from 0 - 32767 Hz).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range of 0..15.
- *milliseconds* is a constant, expression or a bit, nibble, byte or word variable.
- *freq1* and *freq2* are constant, expression or bit, nibble, byte or word variables in the range 0..32767 representing the corresponding frequencies.

GOSUB
GOSUB *addressLabel*
Store the address of the next instruction after GOSUB, then go to the point in the program specified by *addressLabel*.

- *AddressLabel* is a label that specifies where to go.

GOTO
GOTO addressLabel
Go to the point in the program specified by addressLabel.

- *AddressLabel* is a label that specifies where to go.

HIGH
HIGH *pin*
Make the specified pin output high (write 1s to the corresponding bits of both DIRS and OUTS).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

IF...THEN
IF *condition* **THEN** *addressLabel*
Evaluate condition and, if true, go to the point in the program marked by *addressLabel*.

- *Condition* is a statement, such as "x=7" that can be evaluated as true of false.
- *AddressLabel* is a label that specifies where to go in the event that the condition is true.

INPUT
INPUT *pin*
Make the specified pin an input (write a 0 to the corresponding bit of DIRS).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

LOOKDOWN
LOOKDOWN *value*, **{??,} [***value0, value1,... valueN***],** *variable*

- *value* is a constant, expression or a bit, nibble, byte or word variable.

- *??* is =, <>, >, <, <=, =>. (= is the default).
- *value0, value1*, etc. are constants, expressions or bit, nibble, byte or word variables.
- *variable* is a bit, nibble, byte or word variable.

LOOKUP
LOOKUP *index*, **[***value0, value1,... valueN***],** *variable*
Look up the value specified by the index and store it in a variable. If the index exceeds the highest index value of the items in the list, variable is unaffected. A maximum of 256 values can be included in the list.
- *index* is a constant, expression or a bit, nibble, byte or word variable.
- *value0, value1*, etc. are constants, expressions or bit, nibble, byte or word variables.
- *variable* is a bit, nibble, byte or word variable.

LOW
LOW *pin*
Make pin output low (write 1 to the corresponding bit of DIRS and 0 to the corresponding bit of OUTS).
- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

NAP
NAP *period*
Nap for a short period. Power consumption is reduced to 50 µA (assuming no loads).
- *period* is a constant, expression or a bit, nibble, byte or word variable in the range 0..7 representing 18ms intervals.

OUTPUT
OUTPUT *pin*
Make the specified pin an output (write a 1 to the corresponding bit of DIRS).
- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

PAUSE
PAUSE *period*
Pause execution for 0–65535 milliseconds.
- *period* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535.

PULSIN
PULSIN *pin, state, variable*
Measure an input pulse (resolution of 2 µs).
- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *state* is a constant, expression or a bit, nibble, byte or word variable in the range 0..1.

- *variable* is a bit, nibble, byte or word variable.

Measurements are in 2uS intervals and the instruction will time out in 0.13107 seconds.

PULSOUT
PULSOUT *pin, period*
Output a timed pulse by inverting a pin for some time (resolution of 2 µs).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *period* is a constant, expression or a bit, nibble, byte or word
- *variable* in the range 0..65535 representing the pulse width in 2uS units.

PWM
PWM pin, duty, cycles
Output PWM, then return pin to input. This can be used to output analog voltages (0-5V.) using a capacitor and resistor.

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *duty* is a constant, expression or a bit, nibble, byte or word variable in the range 0..255.
- *cycles* is a constant, expression or a bit, nibble, byte or word variable in the range 0..255 representing the number of 1 ms cycles to output.

RANDOM
RANDOM *variable*
Generate a pseudo-random number.

- *variable* is a byte or word variable in the range 0..65535.

RCTIME
RCTIME *pin, state, variable*
Measure an RC charge/discharge time. Can be used to measure potentiometers.

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *state* is a constant, expression or a bit, nibble, byte or word variable in the range 0..1.
- *variable* is a bit, nibble, byte or word variable.

READ
READ *location, variable*
Read EEPROM byte into variable.

- *location* is a constant, expression or a bit, nibble, byte or word variable in the range 0..2047.
- *variable* is a bit, nibble, byte or word variable.

RETURN

Return from subroutine – sends the program back to the address (instruction) immediately following the most recent GOSUB.

REVERSE
REVERSE *pin*
If pin is an output, make it an input. If pin is an input, make it an output.
- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

SERIN
SERIN *rpin***{***fpin***},** *baudmode***, {***plabel***,} {***timeout***,** *tlabel***,} [***inputdata***]**
Serial input with optional qualifiers, time-out, and flow control. If qualifiers are given, then the instruction will wait until they are received before filling variables or continuing to the next instruction. If a time-out value is given, then the instruction will abort after receiving nothing for a given amount of time. Baud rates of 300 - 50,000 are possible (0 - 19,200 with flow control). Data received must be N81 (no parity, 8 data bits, 1 stop bit) or E71 (even parity, 7 data bits, 1 stop bit).
- *rpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..16.
- *fpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *baudmode* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535.
- *plabel* is a label to jump to in case of a parity error.
- *timeout* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535 representing the number of milliseconds to wait for an incoming message.
- *tlabel* is a label to jump to in case of a timeout.
- *inputdata* is a set of constants, expressions and variable names separated by commas and optionally proceeded by the formatters available in the DEBUG command, except the ASC and REP formatters. Additionally, the following formatters are available:

1. STR bytearray\L{\E} input a string into bytearray of length L with optional end-character of E. (0's will fill remaining bytes).
2. SKIP L input and ignore L bytes.
3. WAITSTR bytearray{\L} Wait for bytearray string (ofL length, or terminated by 0 if parameter is not specified and is 6 bytes maximum).
4. WAIT (value {,value...}) Wait for up to a six-byte sequence.

SEROUT
SEROUT *tpin***{***fpin***},** *baudmode***, {***pace***,} {***timeout***,** *tlabel***,} [***outputdata***]**
Send data serially with optional byte pacing and flow control. If a pace value is given, then the instruction will insert a specified delay between each byte sent (pacing is not available with flow control). Baud rates of 300 - 50,000 are possible (0 - 19,200 with flow control). Data is sent as N81 (no parity, 8 data bits, 1 stop bit) or E71 (even parity, 7 data bits, 1 stop bit).
- *tpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..16.
- *fpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *baudmode* is a constant, expression or a bit, nibble, byte or word variable in the range 0..60657.

- *pace* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535 specifying a time (in milliseconds) to delay between transmitted bytes. This value can only be specified if the fpin is not specified.
- *timeout* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535 representing the number of milliseconds to wait for the signal to transmit the message. This value can only be specified if the fpin is specified.
- *tlabel* is a label to jump to in case of a timeout. This can only be specified if the fpin is specified.
- *outputdata* is a set of constants, expressions and variable names separated by commas and optionally proceeded by the formatters available in the DEBUG command.

SHIFTIN
SHIFTIN *dpin, cpin, mode,* **[result{***bits***} { ,***result***{***bits***}... }]**
Shift bits in from parallel-to-serial shift register.

- *dpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the data pin.
- *cpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the clock pin.
- *mode* is a constant, symbol, expression or a bit, nibble, byte or word variable in the range 0..4 specifying the bit order and clock mode. 0 or MSBPRE = msb first, pre-clock, 1 or LSBPRE = lsb first, pre-clock, 2 or MSBPOST = msb first, post-clock, 3 or LSBPOST = lsb first, post-clock.
- *result* is a bit, nibble, byte or word variable where the received data is stored.
- *bits* is a constant, expression or a bit, nibble, byte or word variable in the range 1..16 specifying the number of bits to receive in result. The default is 8.

SHIFTOUT
SHIFTOUT *dpin, cpin, mode***, [***data***{***bits***} {,** *data***{***bits***}... }]**
Shift bits out to serial-to-parallel shift register.

- *dpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the data pin.
- *cpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the clock pin.
- *mode* is a constant, symbol, expression or a bit, nibble, byte or word variable in the range 0..1 specifying the bit order. 0 or LSBFIRST = lsb first, 1 or MSBFIRST = msb first.
- *data* is a constant, expression or a bit, nibble, byte or word variable containing the data to send out.
- *bits* is a constant, expression or a bit, nibble, byte or word variable in the range 1..16 specifying the number of bits of data to send. The default is 8.

SLEEP
SLEEP *seconds*
Sleep for 1-65535 seconds. Power consumption is reduced to approximately 50 µA.

- *seconds* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535 specifying the number of seconds to sleep.

STOP
STOP
Stop the program execution until the power is reset. This command differs from SLEEP in two respects. First, the BASIC Stamp 2 does not enter low power mode when the stop command is executed. Second, outputs are not interrupted every 2.3 seconds.

TOGGLE
TOGGLE *pin*
Invert the state of a pin.

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

WRITE
WRITE *location, data*
Write byte into EEPROM.

- *location* is a constant, expression or a bit, nibble, byte or word variable in the range 0..2047.
- *data* is a constant, expression or a bit, nibble, byte or word variable.

XOUT
XOUT *mpin, zpin*, **[***house******keyorcommand***{***cycles***}** **{,** *house******keyorcommand***{***cycles***}... }]**
Generate X-10 powerline control codes. For use with TW523 or TW513 powerline interface module.

- *mpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the modulation pin.
- *zpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the zero-crossing pin.
- *house* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the house code A..P respectively.
- *keycommand* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying keys 1..16 respectively or is one of the commands shown for X-10 light control in the BASIC Stamp Manual Version 1.9. These commands include lights on, off, dim and bright.
- *cycles* is a constant, expression or a bit, nibble, byte or word variable in the range 2..65535 specifying the number of cycles to send. (Default is 2).

Appendix D: Building Servo Ports on the Rev A Board of Education

Previous to July 2000, Parallax sold the Board of Education Rev A along with Boe-Bot Kits, instead of the Board of Education Rev B. The Board of Education Rev B has four servo ports built into the top of it that makes connecting servos quick and easy. If you have a Board of Education Rev A, building your own servo ports is pretty easy. All the projects in this text were designed so that there would be no conflict with the Rev A servo ports introduced in this appendix. So, if you build the servo ports as instructed here, the experiments in this text should go without a hitch.

The servo ports we are going to make will function as replacements for servo Ports 12 and 13 on the Board of Education Rev B. Once these servo ports are built and tested, you can leave them plugged in every activity from Chapter #1, Activity #6 through the end of the text (Chapter #6, Activity #3). With this in mind, leaving the servo ports on your BOE Rev A connected and not disassembling them between activities is recommended. Any activity that features a schematic with servos connected to I/O pins P12/P13 will work fine with this servo port design shown in Figure D.2. Likewise, whenever a picture shows servos plugged into the BOE Rev B's servo ports 12 and 13, Figure D.2 is the BOE Rev A substitute for the servo port wiring.

Here's how to build the servo ports onto your BOE Rev A:

Parts List:

(1) Board of Education Rev A
(1) 3300µF capacitor
(2) 3-pin headers
(2) Servos
(Misc.) Jumper Wires

The three-pin headers in the Rev A Robotics! Kits are shown in Figure D.1 (a). Modify this three-pin header so that the plastic holding the three pins together is in the middle of the pins as shown in Figure D.1 (b).

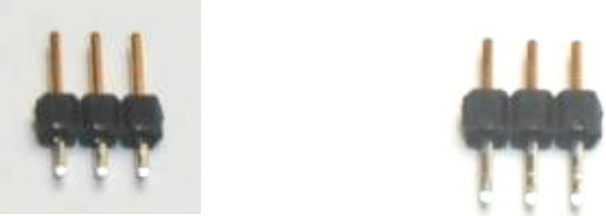


Figure D.1 (a) Unmodified header, and (b) modified header.

Next connect the ports to the Board of Education as shown in Figure D.2.

- Connect the 3300µF capacitor from Vdd to Vss. Make sure the shorter leg of the capacitor is plugged into one of the two sockets labeled Vss on the 20-socket app-mod header. Also make sure that the capacitor's longer pin is plugged into the socket labeled Vdd on the app-mod header.

- Insert the three-pin headers into the breadboard and use jumper wires to connect Vin, Vss, P12 and P13 to the sockets on the white breadboard EXACTLY AS SHOWN in Figure D.2.

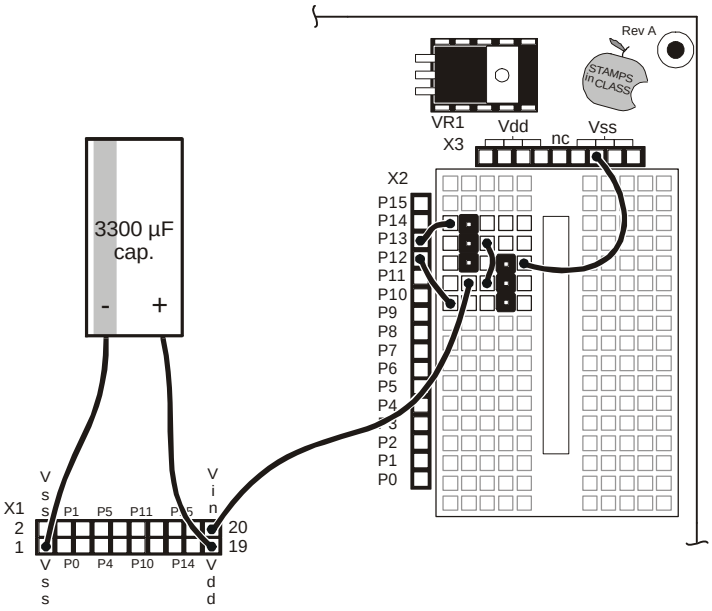


Figure D.2: Servo Port Wiring.

Next, the servos can be connected to the servo ports. The upper three-pin header is equivalent to servo port 13 and the lower three-pin header is equivalent to servo port 12.

Make sure to follow the servo wiring color codes shown in Figure D.3. Failure to do so could result in damaged servos, a damaged BASIC Stamp, or both.

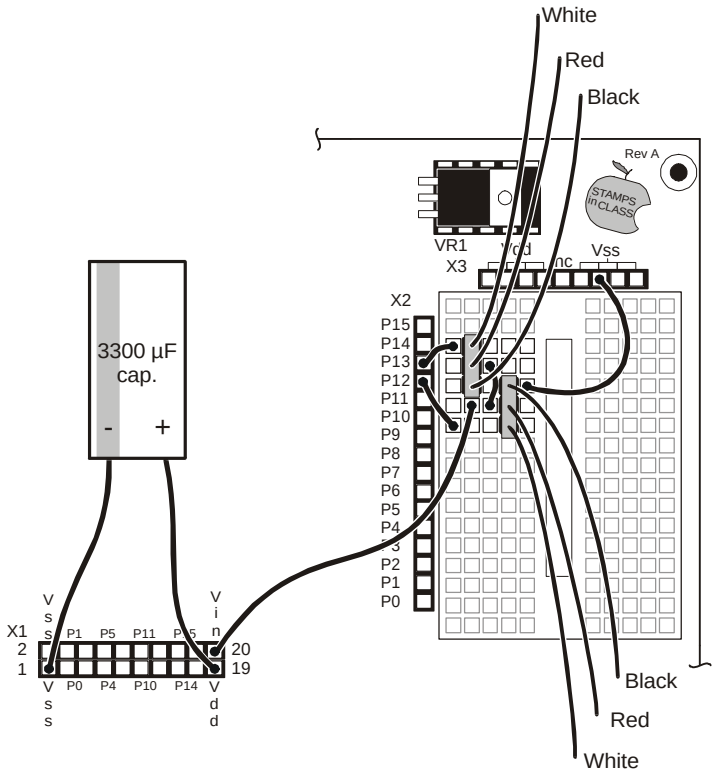


Figure D.3: Color codes for plugging the servos into the servo ports.

Appendix E: Board of Education Rev A Voltage Regulator Upgrade Kit

Prior to June 1999, Parallax produced the Board of Education with a LM7805CV voltage regulator. This voltage regulator is fine for general use, but will cause the BASIC Stamp to reset when used with 6-volt power supplies and higher current devices like servos or stepper motors. If your Board of Education has the LM7805CV voltage regulator, contact Parallax and ask for a free BOE Voltage Regulator Upgrade Kit (stock code #28151). This Appendix includes instructions to remove the LM7805CV voltage regulator and replace it with the low-drop LM2940. Parallax will send it out for free by U.S. Mail.

The voltage regulator is the only component that has a machine screw and nut sticking up on the circuit board located right next to the "Stamps in Class" logo. To put in the new LM2940 regulator, all you need is a small soldering iron, wire cutters, screwdriver and a little solder. Replacing the 7805CV with an LM2940 only takes a few minutes and is well worth the time. Here's how to do it.

Step 1: Remove the 7805 Voltage Regulator

Using your small wire cutter, cut the leads of the existing 7805 regulator off flush with the case of the regulator as shown in Figure E.1. When you're done with this step you should have leads still attached to your circuit board. You don't have to unsolder anything.

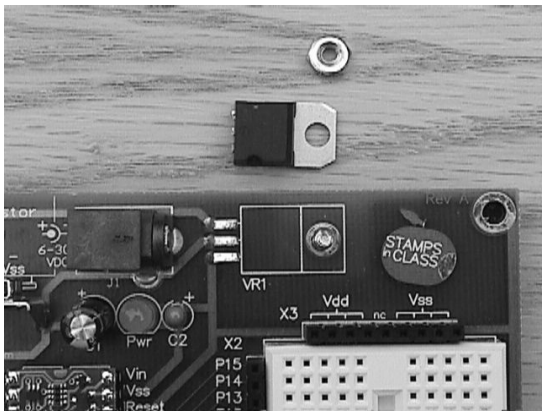


Figure E.1: Remove the 7805 regulator by cutting the leads close to the regulator case and unscrewing the small nut.

Step 2: Install LM2940 Regulator

Put the LM2940 regulator where the old one was, trim the leads off a little and put the screw back in and tighten the nut. Line it up just like the old one was so the leads from the new regulator are laying on top of the ones you left when you cut out the old 7805 regulator. Figure E.2 shows what things should look like at this point.

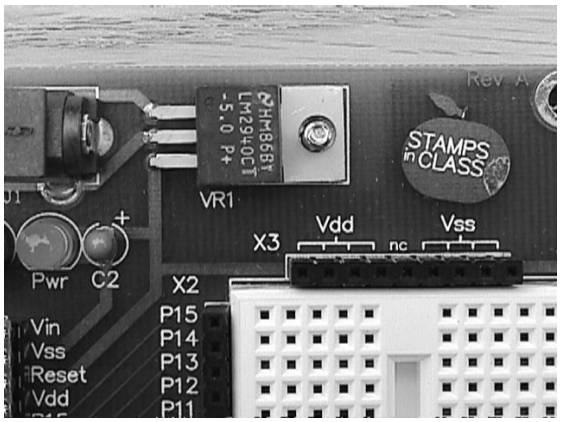


Figure E.2: Put the LM2940 in place of the 7805. The leads will line up with the old leads of the 7805.

Step 3: Solder Regulator on Board of Education

Heat up the soldering iron if you haven't done so yet and get ready to solder the new leads from the LM 2940 onto the old leads underneath them. Before you solder, trim the leads of the LM 2940-5 so they don't extend beyond the old leads. Solder the leads together and you are finished. Be sure you tighten the 4-40 screw and nut that mount the regulator to the Board of Education.

Appendix F: Breadboarding Rules

Figure F.1 shows the circuit symbols used in the experiments along with where to find them on the Board of Education Rev A. The symbol for Vdd is the positive 5-volt supply for the BASIC Stamp and the Board of Education. There are four sockets along the top side of the breadboard to the left for making connections to Vdd.

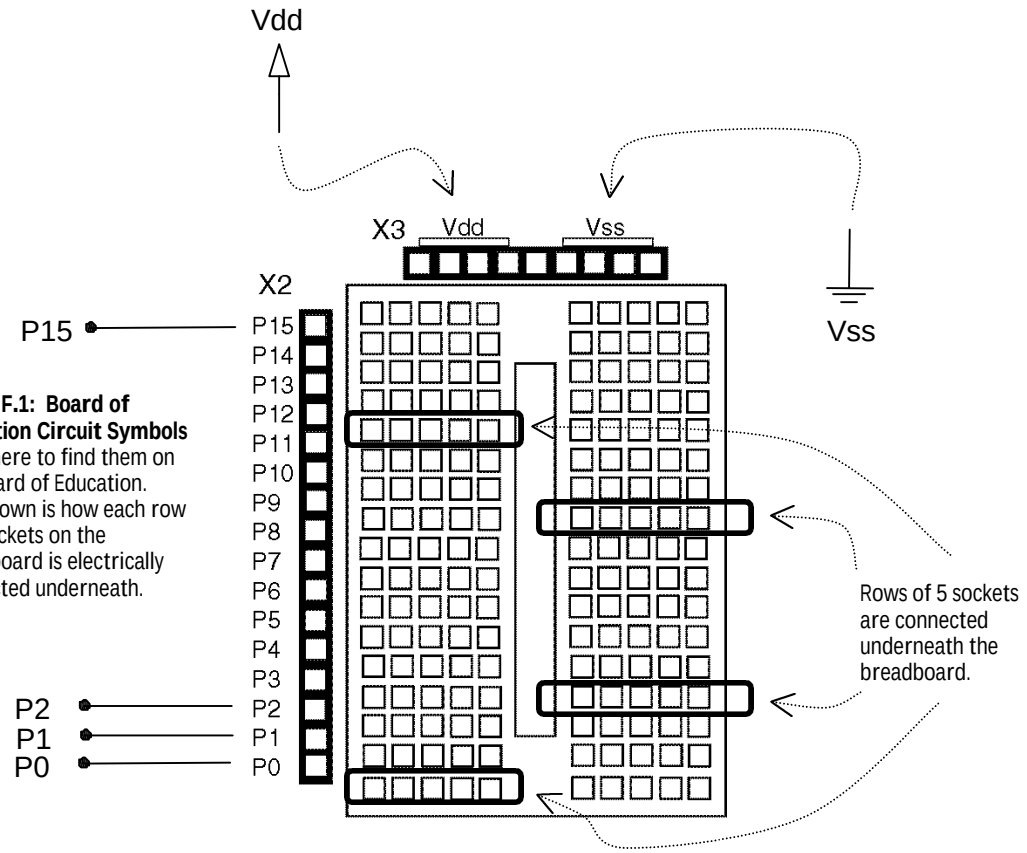


Figure F.1: Board of Education Circuit Symbols and where to find them on the Board of Education. Also shown is how each row of 5 sockets on the breadboard is electrically connected underneath.

Next, the ground symbol is used for Vss. This is the reference terminal for taking measurements, and it's considered to be 0 volts compared to all other voltages on the Board of Education. The four sockets for connecting jumper wires to Vss are along the top of the breadboard to the right.

There is a row of 15 sockets along the left side of the breadboard for connecting to the BASIC Stamp I/O pins. Each I/O pin has a label. I/O pin P0 is connected to the bottom left socket. Pin P1 is the next socket up, and above that socket is the connection to pin P2, and so on through pin P15 at the top left.

Figure F.1 also shows some samples of five-socket-wide rows that are electrically connected underneath the breadboard. There are 34 of these rows arranged in the two columns on the breadboard. If you want to connect two jumper wires to each other, you can just plug them both into the same row of five. Then the wires are electrically connected. Likewise, if you want to connect one or more wires to the terminal of a part, just plug them into the same row on the breadboard and they'll be connected. Four sockets at a time can be added for Vdd, Vin, or Vss by just running a jumper wire from a socket, such as Vdd, to an empty row on the breadboard.

Figure F.2 shows the BOE Rev B breadboard. Along with Vdd and Vss, three sockets are also added for accessing Vin, the BOE's unregulated power source. Whatever voltage is plugged into the BOE is what appears at the Vin sockets. If the power supply is the four 1.5 V AA batteries in the Boe-Bot battery pack, Vin will be at 6 V. If the power supply is a 9 V battery connected to the battery clips on the BOE, Vin will be 9 V. If a wall mount 7.5 V power supply is connected to the BOE's power plug, Vin should be 7.5 V. However, be careful of wall mount power supplies. Their voltage output varies with the current draw placed on the supply.

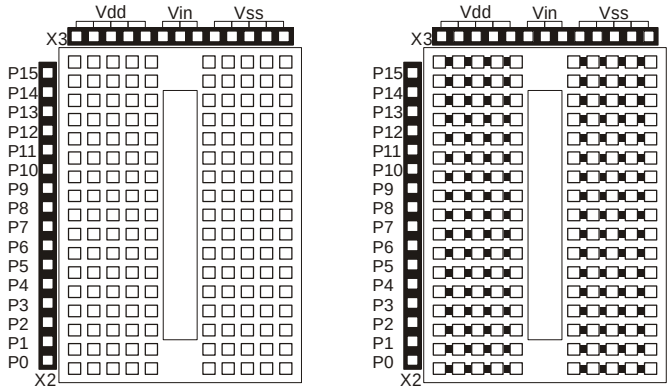


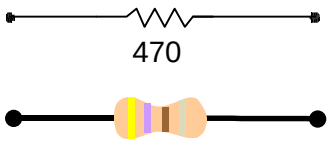
Figure F.2: Another look at the connections underneath the breadboard, this time on the BOE Rev B.

Appendix G: Resistor Color Codes

Figure G.2 shows a drawing of a resistor below its circuit symbol. The circuit symbol typically has the **resistance** value written below or next to it. The colored stripes on the part drawn below the symbol indicate its value, which is measured in **Ohms**. The omega symbol (Ω) is used to denote the Ohm.

Most common types of resistors have colored bands that indicate their value. The resistors that we're using in this series of experiments are typically "1/4 watt, carbon film, with a 5% tolerance." If you look closely at the sequence of bands you'll notice that one of the bands (on an end) is gold. This is band #4, and the gold color designates that it has a 5% tolerance.

Figure G.2: Resistor Circuit Symbol and Corresponding Component



The resistor color code is an industry standard in designating the resistance of a resistor. Each color band represents a number and the order of the color band will represent a number value. The first two color bands indicate a number. The third color band indicates the multiplier (the number of trailing zeros). The fourth band indicates the tolerance of the resistor +/- 5, 10 or 20%.

Table G.1: Variable Declaration Sizes

Color	1st Digit	2nd Digit	Multiplier	Tolerance
black	0	0	1	
brown	1	1	10	
red	2	2	100	
orange	3	3	1,000	
yellow	4	4	10,000	
green	5	5	100,000	
blue	6	6	1,000,000	
violet	7	7	10,000,000	
gray	8	8	100,000.000	
white	9	9	1,000,000,000	
gold				5%
silver				10%
no color				20%

Appendix H: Resistor Color Codes

A resistor has the following color bands:

Band #1. = Red
Band #2. = Violet
Band #3. = Yellow
Band #4. = Gold

Looking at our chart above, we see that red has a value of 2.

So we write: “2.”
Violet has a value of 7.
So we write: “27”

Yellow has a value of 4.
So we write: “27 and four zeros” or “270000.”

This resistor has a value of 270,000 ohms (or 270 kΩ) and a tolerance of 5%.

Appendix H: 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 is 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 1 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

Program Listing H.2 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 Program Listing H.2, but do not disconnect the Boe-Bot from the serial cable.

```
' Robotics! v1.5, Program Listing H.2: IR Frequency Sweep
' {$Stamp bs2}                          ' Stamp Directive.

'----- Declarations --------------

  l_values   var    bit                 ' Store R sensor vals for processing.
  r_values   var    bit                 ' Store L sensor vals for processing.
  IR_freq    var    word                ' Stores L IR freqs from lookup table.

'----- Initialization ------------

  output 2                                     ' Set all I/O lines sending freqout
  output 7                                     ' signals to function as outputs
  output 1
```

```
  freqout 2, 2000, 3000                        ' Declare a variable for counting.
  low 12                                       ' Set P12 and 13 to output-low.
  low 13

'----- Main Routine --------------

main:

  ' Display message to press enter when ready and wait for carriage return.

  debug cr,"Press enter when ready.  ", cr
  serin 16,16468,[wait(cr)]

  ' After carriage return, display table headings.

  debug "IR         left    Right  ", cr
  debug "Freq       Sensor  Sensor ", cr
  debug "--------------------------", cr

  ' Take multiple frequency measurements.

  for ir_freq = 30500 to 46500 step 1000

    check_sensors:                             ' Check Sensors

    l_values = 0
    r_values = 0

    check_left_sensors:
      freqout 7,1,IR_freq
      l_values = ~in8
       pause 10

    check_right_sensors:
      freqout 1,1,IR_freq
      r_values = ~in0
      pause 10

   ' Display each frequency measurement before continuing to the next loop.

    debug dec5 IR_freq, "       ",bin1 l_values, "       ", bin1 r_values, cr

  next

goto main                                      ' Back to main & repeat the process.
```

- Click the upper of the two window panes shown in Figure H.3.

- Press the Enter key. The frequency response data will appear as shown in the figure.

The BASIC Stamp has been programmed to make the Debug Terminal display a "1" if an object was detected and a "0" if an object was not detected. Figure H.3 shows that the left sensor's region of good signal response is between 36500 and 40500. For the left sensor, the frequency response is strong between 37500 and 41500.

- Modify the **`for...next`** loop in Program Listing H.2 so that it steps in increments of 250 and includes the upper and lower limits of both detectors. In the example shown in Figure H.3, the ***`start`***, ***`end`*** and ***`step`*** values of the **`for...next`** loop would be modified as follows:

```
for ir_freq = 36500 to 41500 step 250
```

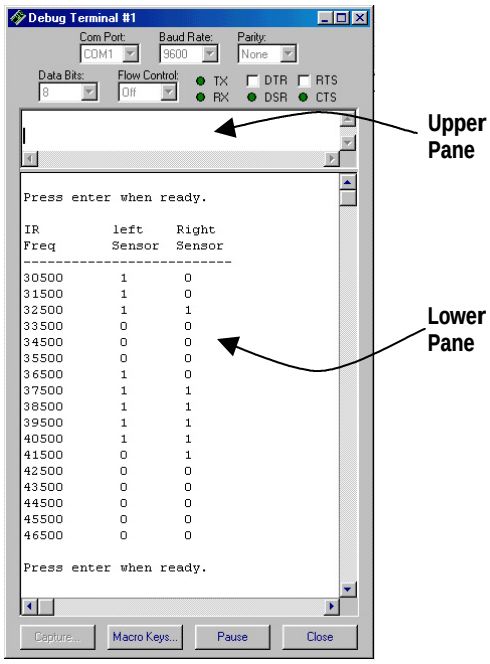


Figure H.3 Debug of Frequency Data.

- Re-run your modified Program Listing H.2, 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 zeros. When the frequency sweep is all zeros, 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 ones 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 ones 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 ones. 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.

How the Frequency Sweep Program Works

Two bit variables, **`l_values`** and **`r_values`**, are declared to temporarily store each IR detector output. A variable named **`IR_freq`** is declared for indexing a **`for…next`** loop.

```
l_values    var       bit
r_values    var       bit
IR_freq     var       word
```

What's

Serial Communication

Serial communication is one of the most common ways for electronic devices to exchange data. Serial communication comes in two flavors, synchronous and asynchronous.

The most common form of synchronous serial communication features a clock line and a data line. When the device sending the data has made a new binary value available at its data output, it sends a pulse via the clock output signaling that the data is ready.

Asynchronous serial communication can be accomplished with just one line for data transmission. Both the device sending and the device receiving the information know at what speed the data will be transmitted. Commonly referred to as the baud rate, this speed is measured in bits per second or bps.

Asynchronous serial messages also have a start bit, a fixed number of data bits, a stop bit, and sometimes a parity bit. The stop bit is a binary value that signifies the resting state between data packets. When the binary value on the data line changes from the resting state, it signifies a start bit. Since both the transmitting device and the receiving device know the baud rate, the transmitter knows exactly how long to send the start bit for before sending the first data bit. Likewise, the receiving device knows exactly how long to wait after the start bit before recording the first data bit.

The BASIC Stamp Manual does a great job of explaining the finer points of both synchronous and asynchronous serial communication. For more information about how synchronous serial communication works, consult the **shiftin** and **shiftout** commands. For more information about how asynchronous serial communication works, consult the **serin** and **serout** commands.

The first debug command in the main routine is nothing new. It displays a message for the user to press the enter key when ready (to run a frequency sweep). The three debug statements that follow are nothing new either. However, the command, **serin 16,16468,[wait(cr)]** is new. **Serin** is a multipurpose command designed to receive asynchronous serial messages from other electronic devices.

```
debug cr,"Press enter when ready.  ", cr
serin 16,16468,[wait(cr)]

debug "IR         left    Right  ", cr
debug "Freq       Sensor  Sensor ", cr
debug "--------------------------", cr
```

The BASIC Stamp uses this particular **serin** command to listen for a serial message from the PC on I/O pin P16, which is the Sin line connected to the computer's data transmit (DTR) line via the serial cable. The ***baudmode*** argument is 16468, which was selected from a table in the **Serin** command section of the BASIC Stamp Manual. The value 16468 tells the BASIC Stamp to communicate with the Debug Terminal. The specification for the Debug Terminal is 9600 bits per second (bps), 8-bits, no parity, one stop bit. These arguments are explained in detail in the BASIC Stamp Manual.

It's most common for **serin** to feature a variable between the square brackets. This way, data sent from the PC to the BASIC Stamp can be saved and used by the application. In this case, the argument **wait(cr)** does not save any data. All it does is tell the **serin** command to wait for a carriage return. The **serin** command does nothing until the carriage return is received. When the **serin** command receives the carriage return, the program moves on to the next command. In other words, **serin** allows the program to continue running after it receives the carriage return. The result of this command is that you can take your time positioning the Boe-Bot in front of its paper target, then hit the carriage return when you're ready for more frequency sweep data.

Before it's modified, the **for...next** loop sweeps the value of **IR_freq** from 30500 to 46500 in steps of 1000. The **freqout** commands within the **for...next** loop use the value of **IR_freq** to set their ***frequency*** arguments.

As with the previous program examples, the value of the IR detector's output is inverted using the NOT operator (~), then stored in a bit variable, such as **l_values** for the left IR pair.

```
for ir_freq = 30500 to 46500 step 1000

    check_left_sensors:
      freqout 7,1,IR_freq
      l_values = ~in8
      pause 10


    check_right_sensors:
      freqout 1,1,IR_freq
      r_values = ~in0
      pause 10
```

Next, the **debug** output values are printed in a format that displays as a table in the Debug Terminal. To make it display properly, blank spaces are inserted between each value using the **" "** message. The width of each number is also controlled by formatters such as **bin5** and **dec1**.

```
debug dec5 IR_freq, "      ",bin1 l_values, "      ", bin1 r_values, cr

next
```

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 I: Boe-Bot Competition Maze Rules

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 it can do. The robot's controller 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 Rules

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.

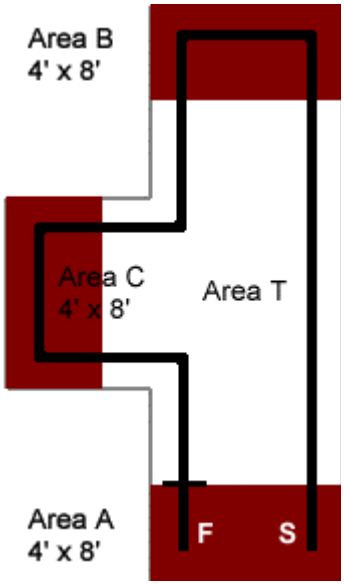
Skills Tested:

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 3/4 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 Tyvec. Tyvec 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.

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.

Accomplishments	Reduction
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.

Appendix I: Boe-Bot Maze Competition Rules

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.